Courtage sémantique de services de calcul

Aurélie Hurault

▶ **To cite this version:**

Aurélie Hurault. Courtage sémantique de services de calcul. Génie logiciel [cs.SE]. Institut National Polytechnique de Toulouse - INPT, 2006. Français. NNT : . tel-00483894

HAL Id: tel-00483894

https://theses.hal.science/tel-00483894

Submitted on 17 May 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre : 2415 Année 2006

THÈSE

présentée pour obtenir le titre de

DOCTEUR DE L'INSTITUT NATIONAL
POLYTECHNIQUE DE TOULOUSE

École doctorale : Informatique et Télécommunications
Spécialité : Informatique

Courtage sémantique de services de calcul

Aurélie HURAULT

Soutenue publiquement le 4 décembre 2006 devant un jury composé de :

MME.	Thérèse HARDIN	Professeur, Université Pierre et Marie Curie - LIP6	Présidente de Jury
M.	Michel DAYDÉ	Professeur, INPT-IRIT	Directeur de thèse
MM.	Michel BIDOIT	Directeur de recherche CNRS, LSV	Rapporteurs
	Thierry PRIOL	Directeur de recherche INRIA, IRISA	
MM.	Frédéric DESPREZ	Directeur de recherche INRIA, LIP	Examinateur
	Marc PANTEL	Maître de conférences, INPT-IRIT	Co-encadrant

Pour toi,
parti alors que je posais ces premiers mots sur le papier.

Remerciements

En avant propos de ce mémoire, je tiens à remercier Messieurs Michel Bidoit et Thierry Priol d'avoir accepté la lourde charge de rapporteur. Je remercie également Madame Thérèse Hardin et Monsieur Frédéric Desprez de l'intérêt qu'ils ont porté à mon travail en acceptant de participer à ce jury.
Je n'oublie pas Monsieur Michel Daydé qui a accepté d'être mon directeur de thèse et qui a toujours été présent quand je le sollicitais.
J'adresse des remerciements particuliers à Monsieur Marc Pantel pour avoir encadré ce travail, pour son investissement, pour avoir répondu présent quand il le fallait et pour avoir su gérer mes coups de stress assez fréquents et pas toujours justifiés !
Je remercie également Messieurs Alain Ayache et Patrick Sallé pour m'avoir permis de rejoindre l'équipe des enseignants du département d'Informatique et de Mathématiques Appliquées en tant que monitrice puis en tant qu'ATER.

Je tiens également à exprimer ma reconnaissance à Pascaline, pour avoir relu ce manuscrit, mais surtout pour son soutien tout au long de cette thèse, que ce soit en rapport avec la recherche, l'enseignement ou plus personnel. Pour les bons moments aussi, merci.
Je remercie également Philippe Quéinnec pour le temps qu'il a accordé à la relecture d'une partie de ce manuscrit, tous les collègues vers qui je me suis tournée un jour pour en discuter ainsi que mes amis et collègues doctorants.
J'ai une pensée particulière pour mes collègues de bureau : Ahmed, Carlos et Pierre-Loïc pour les heures passées en leur compagnie.
Je remercie également tous les résidents du quatrième étage et l'ensemble du personnel du laboratoire pour leur accueil et ces années passées en leur compagnie.

Je n'oublie bien sûr pas ceux qui n'ont pas un rapport direct avec le travail de ce manuscrit, mais sans qui je n'en serais sûrement pas là aujourd'hui.
En premier lieu, je pense à ma famille qui m'a toujours soutenue et encouragée dans mon envie de réaliser ce projet, toujours présente à mes côtés malgré la distance, un soutien sans faille et précieux.
Je pense également à Gwen, bientôt 15 ans d'une amitié solide qui m'a fait évoluer ; à Eléonore, Sandra, Marie-Luce et Soizic, rencontrées lors de mes tous premiers jours à Toulouse et sans qui ces années dans le sud n'auraient pas été les mêmes. Bienvenue à Mathys.
Je n'oublie pas les copines de SB pour les discussions rarement sérieuses et toujours drôles !
Merci également à mes amies de l'équitation et à mes « gros » pour ma bouffée d'air du vendredi soir. Je finirai en remerciant les nouveaux arrivés dans ma vie qui ont égayé cette fin de thèse.

Résumé

La recherche du ou des services de calcul scientifique disponibles sur une grille qui répondent aux besoins d'un utilisateur, aussi appelée courtage de services, est une activité complexe. En effet, les services disponibles sont souvent conçus pour répondre de manière efficace à de nombreux besoins différents. Ceux-ci comportent donc en général de nombreux paramètres et la simple signature du service ne suffit pas pour que l'utilisateur puisse le trouver.

La solution proposée dans ces travaux consiste à utiliser une description formelle du domaine d'application comportant l'ensemble des données et des opérateurs du domaine ainsi que les propriétés des opérateurs. Dans le cadre de cette thèse, cette description est effectuée sous la forme d'une spécification algébrique. Un service ou une requête sont alors des termes de l'algèbre associée. En ce qui concerne les signatures, nous combinons le sous-typage des sortes et la surcharge des opérateurs selon le système de type proposé par G. Castagna pour le $\lambda\&$-calcul. Le courtage consiste alors à effectuer un filtrage modulo la théorie équationnelle associée à la spécification, entre le terme représentant le service souhaité et les termes représentant les services disponibles.

Nous proposons et avons implanté deux algorithmes différents inspirés d'un parcours de l'arbre de recherche des solutions contraint par une quantité d'énergie (nombre d'équations et/ou de compositions applicables). Le premier algorithme est directement dérivé des travaux de Gallier et Snyder sur l'unification équationnelle. Nous avons montré sa correction et argumenté sa complétude (ou exhaustivité). Le second algorithme découle d'une définition constructive de l'ensemble des services qui peuvent répondre à la requête d'un utilisateur. L'algorithme consiste alors en un parcours particulier de l'arbre construit pour engendrer le service requis. Nous avons également montré sa correction, et sa complétude pour certaines formes d'équations.

Nous illustrons notre approche dans les domaines applicatifs suivants : algèbre linéaire et optimisation, et nous nous intéressons au traitement de la combinaison de domaines applicatifs.

Mots clés Courtage de services, Filtrage équationnel, Spécification algébriques, Grille.

Abstract The search for a computing service available on a grid which corresponds to an user's needs, also called trading services, is a complex activity. Indeed available services are often implemented to fulfill efficiently different requirements.These services offer many parameters and their signature is often not enough for a precise description.

The object of this work consists in using a formal description of the dedicated domain : the data, the operators and their properties. This description is carried out as an algebraic specification. Services and requests are then terms of the associated algebra. For the

signatures, we combine sub-sorting and overloading of operators by using the type system proposed by G. Castagna for the $\lambda\&$-calculus. Trading is then implemented as an equational matching modulo the theory associated to the specification, between the term representing the required service and the terms representing the available services.

We propose two algorithms inspired from a traversal of the solution search tree constrained by an amount of energy. The first one is directly inspired from the work of Gallier and Snyder on equational unification. We have proved its correctness and argued its completeness. The second one is derived from a constructive definition of the set of services which answer the user request. We have proved its correctness and its completeness for some kind of equations.

We illustrate our approach with applications to linear algebra and optimization and an interaction between these two domains.

Keywords Trading, Equational matching, Algebraic specification, Grid

Table des matières

Chapitre 1

Introduction

1.1 La problématique

Nous assistons depuis une dizaine d'années à une évolution des techniques informatiques qui conduit à une globalisation des ressources grâce à leur mise à disposition sur le réseau mondial. L'accès aux ressources devient ainsi semblable aux réseaux d'électricité ou d'eau, avec les notions de « Computational Grid », « Data or Storage Grid » ou « Service Grid » (grille de calcul, de stockage ou de service). Ces ressources peuvent aussi bien être des données, des services (sous forme de composants logiciels, de fonctions dans une bibliothèque téléchargeable ou de services exécutables à distance avec échange des données et des résultats, ...), des machines, voire des équipements de mesures (microscope électronique, ...) ou d'interactions (réalité augmentée, ...). Cette disponibilité de ressources diverses introduit le besoin de mécanismes permettant de trouver les ressources répondant le mieux aux besoins d'un client. Ces mécanismes sont appelés courtage de ressources.

Deux types de préoccupations peuvent être pris en compte : d'une part les aspects fonctionnels et d'autre part les performances ou aspects non fonctionnels. Dans le cas des aspects purement fonctionnels, nous nous retrouvons dans le cadre de la recherche de fonctions dans une bibliothèque. Dans le cas non fonctionnel, les préoccupations peuvent concerner les performances, le volume mémoire nécessaire, la qualité de service, le temps de calcul, le débit et la latence de communication, l'espace de stockage, ... Des intergiciels de grille comme Globus [1] [FK97], XtremWeb[FGNC01], NetSolve [AAB+01], DIET [CDL+02], NINF [TNS+03], ITBL [FSYH03] ou encore CONDOR [TTL05] participant à la résolution des problèmes de cette seconde catégorie.

Dans certains domaines, comme le calcul scientifique ou le traitement d'un grand volume de données, les deux aspects sont très importants. Notre travail se place dans le cadre de ces domaines. Nous nous intéressons plus particulièrement au courtage fonctionnel, dans le but de l'intégrer par la suite aux intergiciels de grille, ce qui permettra de coupler un courtage fonctionnel précis et un courtage non fonctionnel performant.

Le courtage fonctionnel associe à chaque composant une description de sa sémantique, c'est-à-dire des fonctionnalités rendues, puis confronte la description du service recherché à l'ensemble des descriptions des services disponibles. Les technologies actuelles de courtage reposent sur le nom du service, l'équivalence entre les signatures des opérations (types des paramètres et des résultats), des mots-clés et/ou des ontologies. Malheureusement, ces solutions ne permettent actuellement pas, soit d'exprimer des informations

[1] `http://www.globus.org/`

suffisamment précises sur le calcul effectué par le service, soit de disposer d'une comparaison entre les descriptions en temps fini.

L'objectif principal de nos travaux est de répondre à ces difficultés. Nous proposons pour cela d'utiliser une description de cette sémantique inspirée des techniques formelles de spécification et de développement du logiciel. Un problème majeur se pose dans le cas général : soit la sémantique est très pauvre, soit la comparaison n'est pas complète ou décidable (les algorithmes de comparaisons peuvent ne pas terminer). Si la comparaison est semi-décidable, il existe alors un algorithme qui donne une réponse positive en temps fini en cas d'égalité et dont l'exécution peut être infinie en cas d'inégalité. Dans le cas de la semi-décidabilité, nous pouvons envisager de borner le temps attribué à l'algorithme de comparaison. Nous comparons alors les services disponibles avec la requête de l'utilisateur. Si la réponse est obtenue dans ce délai, nous la transmettons, sinon nous concluons que le service ne répond pas aux besoins. En augmentant cette borne, nous pouvons obtenir successivement les différents services qui répondent à la requête. En donnant la maîtrise de cette borne à l'utilisateur, celui-ci pourra l'augmenter jusqu'à être satisfait ou décider de s'arrêter. Ces procédures de comparaison sont souvent coûteuses. La restriction à un domaine d'applications limité permet généralement de réduire cette complexité.

L'originalité de l'approche que nous proposons consiste donc, d'une part, à utiliser une sémantique précise disposant de procédures de comparaison semi-décidables, et d'autre part, à se placer dans des domaines d'application spécifiques pour disposer d'un nombre d'opérations restreint et de meilleurs temps de réponse. Les informations sémantiques seront fournies par des spécialistes du domaine applicatif a priori non spécialistes du formalisme choisi pour exprimer la sémantique. Il faut donc que le traitement des informations soit totalement automatique.

L'objectif de ces travaux est donc l'étude de l'apport d'une description de la sémantique des services issue des techniques de spécifications formelles dans le cadre du courtage de services dans des domaines d'application restreints.

De plus, nous ne voulons pas seulement être en mesure d'indiquer les services qui permettent de résoudre un problème fonctionnel, mais nous voulons aussi donner les valeurs que doivent prendre les paramètres pour répondre au problème posé. Nous voulons également pouvoir renvoyer les compositions de services qui résolvent le problème.

Le projet TLSE[2] [DGH+04, CDL+06, CDD+05, DHP05] financé par l'ACI GRID[3] comporte les composants PRUNE et WEAVER de description et de courtage de services et de données sur une grille. Celui-ci nous fournit un premier sujet d'étude en algèbre linéaire creuse. Cette approche est ensuite étendue à d'autres domaines dont la sémantique peut être décrite précisément et simplement par un langage d'expressions mathématiques. Actuellement, seule l'optimisation a été traitée, mais d'autres domaines comme les statistiques et probabilité, le traitement de l'image ou le traitement du signal sont en cours d'étude.

De cette analyse rapide émergent les besoins suivants :

R1 Les procédures de courtage (comparaison de la requête et des services) doivent être semi-décidables.

R2 La description d'un domaine d'application doit être réalisable par un spécialiste de ce domaine qui n'est pas spécialiste du formalisme choisi.

R3 Le traitement de cette description doit être complètement automatisé.

[2] `http://www.enseeiht.fr/lima/tlse/index.html`

[3] `http://www-sop.inria.fr/aci/grid/public/acigrid.htm`

R4 La description doit être assez précise pour que la comparaison puisse permettre l'extraction des valeurs des paramètres des services.

R5 La méthode de comparaison doit répondre à des contraintes de performance, nous devons donc pouvoir implanter des heuristiques d'accélération de l'algorithme.

Dans la suite de ce chapitre, différentes descriptions possibles et les méthodes de comparaison associées vont être abordées. Nous identifierons celles qui répondent à nos besoins et expliquerons pourquoi notre choix s'est porté sur les spécifications formelles et, en particulier, sur les spécifications algébriques. Nous détaillerons ensuite différents projets avec lesquels nous avons des préoccupations communes et comment celles-ci peuvent être prises en compte.

Dans le deuxième chapitre, nous reviendrons plus en détail sur les spécifications algébriques et formaliserons notre problème. Les troisième et quatrième chapitres détailleront deux prototypes qui ont été réalisés pour répondre à notre problématique. Le premier est basé sur les travaux sur l'unification équationnelle de Gallier et Snyder [GS89] et le second est dérivé de notre formalisation du problème. Le premier traite les différents sous-problèmes engendrés à partir du problème initial de façon totalement indépendante et parallèle, alors que le second réalise un traitement en série. Cela implique que le premier est facilement parallélisable, mais que le second détecte les erreurs de façon plus précoce et est donc plus efficace algorithmiquement. La correction des deux algorithmes sera montrée ainsi que la complétude du second dans le cas de théories équationnelles particulières. Nous illustrerons ensuite les possibilités de notre approche sur différents exemples issus de deux domaines (algèbre linéaire et optimisation). Nous conclurons par un bilan et l'exposition de perspectives.

1.2 Descriptions possibles et méthodes de comparaison associées

Différentes approches existent pour la description et le courtage de services. Une étude pour la réutilisation de code a été effectuée dans [MMM98]. Dans ce chapitre, nous reprendrons certaines des descriptions étudiées dans cet article, mais nous en décrirons également d'autres. En effet, cet article date de 1998 et de nouvelles techniques ont vu le jour depuis. Notamment, l'apparition des web services a fait naître de nouveaux besoins et de nouvelles solutions.

Un web service est un composant réalisé dans un langage quelconque, déployé sur une plate-forme quelconque et enveloppé dans une couche standard exprimée en XML. Il peut être découvert et invoqué dynamiquement par d'autres services. Afin que cette découverte et cet appel soient réalisables, une description doit être ajoutée à ce web service. L'ensemble s'intègre dans le web sémantique. Cette description se fait initialement à l'aide de WSDL[4] (Web Services Description Language) qui est une approche basée sur les signatures, et/ou RDF[5] (Resource Description Framework) [LS99] et UDDI[6] qui sont des approches par mots-clés. Pour pallier le manque de précision de ces descriptions, des ontologies ont été développées pour exprimer la signification des mots-clés et des relations

[4] `http://www.w3.org/TR/wsdl`

[5] `http://www.w3.org/RDF/`

[6] `http://www.uddi.org`

qui les lient. DAML+OIL[7] et OWL[8] [MvH04] sont des langages utilisés pour décrire ces ontologies.

Nous retrouvons cette approche par signature dans les approches classiques des langages de programmation, des composants, des types abstraits, des modules, ... Cette information peut également être enrichie de mots-clés pour apporter une information complémentaire.

Il existe d'autres descriptions plus formelles comme les spécifications algébriques, la logique de Hoare, qui permet d'exprimer la sémantique par des relations logiques entre les paramètres et les résultats. Nous nous intéresserons plus particulièrement à la description par spécifications algébriques, qui est celle que nous avons adoptée.

Les différentes descriptions seront développées sur l'exemple suivant, où les lettres majuscules désignent des matrices et les lettres grecques des scalaires. Les services disponibles seront :

- $serv_1(matrix\ A, matrix\ B) = A + B$
- $serv_2(matrix\ A, matrix\ B) = A * B$
- $serv_3(matrix\ A, matrix\ B, matrix\ C, matrix\ D) = A * B + C * D$
- $serv_4(real\ \alpha, matrix\ A) = \alpha * A$

1.2.1 Les signatures et isomorphismes de types

Les caractéristiques des services les plus faciles à obtenir sont le nombre et le type des entrées (paramètres) et sorties (résultats). La plupart des langages de programmation proposent de définir les signatures des méthodes dans une interface. En Corba[9], par exemple, les interfaces sont exprimées en IDL[10] (Interface Definition Language) et des mécanismes d'introspection d'interfaces (DII et DSI [GGM99]) permettent la découverte des signatures des services disponibles. La signature des services est une information importante que doit contenir la description d'un service. En effet, si le service est invoqué avec des types incompatibles avec la signature, il y aura une erreur.

1.2.1.1 Technique de description : IDL (CORBA)

En IDL, l'interface décrivant les services de notre exemple s'écrit de la façon suivante :

```
matrix serv1(in matrix A, in matrix B) ;
matrix serv2(in matrix A, in matrix B) ;
matrix serv3(in matrix A, in matrix B,
             in matrix C, in matrix D) ;
matrix serv4(in real alpha, in matrix A) ;
```

Notons que les types ne sont pas suffisants pour distinguer serv1 et serv2.

[7] `http://www.daml.org/2000/12/reference.html`
[8] `http://www.w3.org/2004/OWL/`
[9] `http://www.omg.org/corba/`
[10] `http://www.omg.org/gettingstarted/omg_idl.htm`

1.2.1.2 Technique de description : WSDL (Web Services)

Dans le cas des web services, un fichier WSDL (Web Services Description Language) est utilisé pour décrire les types des paramètres d'entrée et de sortie. Le fichier suivant est un extrait du fichier WSDL engendré à l'aide d'Axis[11] pour décrire les entrées et les sorties des services 1 et 2. Pour simplifier le fichier et ne pas avoir à gérer le mapping des données, les matrices ont été remplacées par des entiers.

```
<wsdl:message name="serv2Request">
  <wsdl:part name="A" type="xsd:int"/>
  <wsdl:part name="B" type="xsd:int"/>
</wsdl:message>
<wsdl:message name="serv1Response">
  <wsdl:part name="serv1Return" type="xsd:int"/>
</wsdl:message>
<wsdl:message name="serv2Response">
 <wsdl:part name="serv2Return" type="xsd:int"/>
</wsdl:message>
<wsdl:message name="serv1Request">
  <wsdl:part name="A" type="xsd:int"/>
  <wsdl:part name="B" type="xsd:int"/>
</wsdl:message>
<wsdl:portType name="ExempleThese">
  <wsdl:operation name="serv1" parameterOrder="A B">
    <wsdl:input message="impl:serv1Request"
                name="serv1Request"/>
    <wsdl:output message="impl:serv1Response"
                 name="serv1Response"/>
  </wsdl:operation>
  <wsdl:operation name="serv2" parameterOrder="A B">
    <wsdl:input message="impl:serv2Request"
                name="serv2Request"/>
    <wsdl:output message="impl:serv2Response"
                 name="serv2Response"/>
  </wsdl:operation>
</wsdl:portType>
...
```

Les noms des services et des paramètres sont présents dans des balises de la forme :

```
<wsdl:operation name="NOM" parameterOrder="P1 ... Pn">
```

Les types des paramètres sont présents dans des balises de la forme :

```
<wsdl:message name="NOMRequest">
  <wsdl:part name="P1" type="T1"/>
...
  <wsdl:part name="PN" type="TN"/>
</wsdl:message>
```

Le type de retour est présent dans une balise de la forme :

```
<wsdl:message name="NOMResponse">
  <wsdl:part name="NOMReturn" type="T"/>
</wsdl:message>
```

[11] `http://ws.apache.org/axis/`

Les types possibles sont ceux définis à l'aide des balises « simpleType » et « complexType » de XSD (XML Schema Definition)[12].

1.2.1.3 Technique de comparaison : Isomorphismes de type

De nombreux travaux ont été réalisés pour trouver un nom de service dans une librairie à partir de la signature du service recherché. Citons par exemple [Rit89, RT89, ZW93].

Ces travaux soulignent l'importance de ne pas se limiter à une correspondance exacte des types. En effet, la fonctionnalité du service est la même si l'ordre des paramètres du service est permuté. Mais, il peut également être intéressant de proposer des services dont la signature se rapproche de celle recherchée sans être, pour autant, exactement identique. Dans le cas de systèmes polymorphes, il peut aussi être intéressant de renvoyer des signatures plus générales que la requête.

Les travaux existants permettent par exemple de retrouver le service 4 à partir des recherches des signatures suivantes :

- $(real, matrix) \rightarrow matrix$
- $(matrix, real) \rightarrow matrix$
- $matrix \rightarrow (real \rightarrow matrix)$
- $real \rightarrow (matrix \rightarrow matrix)$
- ...

Dans [ZW93], une procédure pour rechercher des modules représentés sous la forme d'un ensemble de services est proposée. Elle est, elle aussi, relaxée pour pouvoir trouver une sous partie du module.

Dans [DPR05], les auteurs proposent de retrouver une interface dans le contexte d'un langage objet. Ils se placent donc dans le cadre de types récursifs avec sous-typage et d'un produit cartésien associatif et commutatif.

Cette recherche des types isomorphes est réalisée à partir d'un ensemble figé d'équations sur les types. Des propriétés telles que la commutativité dans les paires pourront être décrites, mais si un nouveau type est ajouté (comme les triplets), l'intégralité de l'algorithme de comparaison doit être reconstruit.

1.2.1.4 Limites de l'approche

Il est évident que ces mécanismes peuvent limiter l'espace de recherche d'un service, mais ils ne permettent pas de répondre entièrement à notre problème.

La raison la plus évidente est qu'ils ne distinguent pas deux services qui ont la même signature. Par exemple, il sera impossible de distinguer une addition (serv1) d'une multiplication (serv2) ce qui est, dans notre contexte, un obstacle majeur à l'utilisation d'une telle description et méthode de comparaison.

De plus, si notre requête est $req(matrix\ X, matrix\ Y, matrix\ Z) = X * (Y + Z)$, aucune correspondance ne sera trouvée alors que la composition des services serv1 et serv2 ou un choix judicieux des paramètres de serv3 permettraient de répondre à cette requête.

Néanmoins, cette description n'est pas écartée car, enrichie d'autres informations, elle peut permettre de réduire l'espace de recherche. Mais, à elle seule, elle n'est pas assez expressive pour pouvoir répondre à nos objectifs.

[12] `http://www.w3.org/TR/xmlschema-1/`

Dans l'approche que nous avons choisie, la comparaison n'est pas effectuée en faisant une restriction de l'espace de recherche grâce aux types suivie d'une comparaison plus poussée, pour plusieurs raisons :

- Le fait que nous cherchions également les compositions de services implique qu'un service peut partiellement répondre à notre requête, il n'est donc pas pertinent de rechercher la signature complète.
- Nous acceptons des signatures beaucoup plus complexes que celle que nous cherchons, car nous autorisons que des paramètres qui n'agissent pas sur l'aspect fonctionnel du service soient présents dans la signature. Nous ne saurons pas les déterminer, mais nous ne rejetons pas le service pour autant.
- L'ordre des paramètres n'intervient pas dans notre algorithme, puisque c'est la fonctionnalité en elle-même qui est décrite. C'est le nom du paramètre et son rôle qui est important et non sa position dans la signature.

1.2.2 Exploitation de mots-clés

Le problème de l'approche par signatures étant l'absence de renseignements sur le lien entre les paramètres en entrée et les résultats en sortie, cette approche peut être combinée avec une description à l'aide de mots-clés, qui permet d'ajouter l'information manquante et ainsi de se rapprocher au plus près de la sémantique réelle du service.

La description par mots-clés peut prendre différents aspects pour un service donné : un seul mot-clé, une liste de mots-clés, une unique association attribut / mot(s)-clé(s) ou une liste d'associations attribut / mot(s)-clé(s).

L'avantage de cette approche est que la comparaison de la description qui en découle est simple (égalité entre mots-clés ou inclusion d'une liste de mots-clés dans une autre).

1.2.2.1 Technique de description : Corba Trading Service

Cette approche par mots-clés est utilisée dans Corba Trading Service. Le *trader service* est un service de pages jaunes. Il permet de décrire les services par une interface définissant les opérations et les attributs pouvant être utilisés et une liste de propriétés décrivant le service sous la forme de séquences de paires (*nom de l'attribut*,*valeur*). La recherche d'un service s'effectue alors en indiquant les valeurs souhaitées pour les propriétés. Dans notre exemple, nous choisissons de décrire les services à l'aide d'un nom plus explicite que $serv_i$.

```
serv1.nom("matrix-addition");
serv2.nom("matrix-matrix-multiplication");
serv3.nom("matrix-matrix-addition-multiplication");
serv4.nom("scalar-matrix-multiplication");
```

L'addition et la multiplication peuvent maintenant être différenciées, mais la connaissance du nom d'un service est requise pour l'utiliser. Il est donc nécessaire soit de le connaître, soit de le deviner.

Nous pourrions ajouter une propriété « description » qui permettrait de décrire plus précisément le service, mais la façon de décrire un même service est différente selon les personnes. L'interprétation sémantique du vocabulaire peut différer et même la langue de la description peut varier. De plus, s'il paraît envisageable de décrire les deux premiers services en langage naturel, il sera plus difficile de décrire le troisième de manière unique et non ambiguë.

1.2.2.2 Technique de description : RDF

RDF est un standard de description de méta-données. Cette description est écrite sous la forme d'un document XML. Comme le service de courtage de Corba, elle est basée sur des paires (*nom de l'attribut, valeur*). Elle en présente donc les inconvénients. Néanmoins, la forme de la valeur peut être plus complexe qu'une chaîne de caractères, ce qui laisse une plus grande liberté. L'exemple suivant propose une description du premier service.

```
<rdf:Description rdf about="serv1">
  <p:parameters>
    <rdf:Seq>
     <rdf:li nom="A" type="Matrix" />
     <rdf:li nom="B" type="Matrix" />
    </rdf:Seq>
  </p:parameters>
  <r:result>Matrix></r:result>
  <n:nom>addition</n:nom>
  <s:specification>A+B</s:specification>
</rdf:Description>
```

La valeur d'un attribut peut contenir des balises XML. Dans ce cas il faut indiquer à l'analyseur RDF de ne pas interpréter le balisage mais de le retenir comme faisant partie de la valeur.

Dans l'exemple donné, la spécification sous la forme d'une chaîne de caractères « A+B » pourrait alors être représentée par un terme MATHML[13] ou OpenMath[14]. Le champ valeur pouvant prendre une forme quelconque, nous pourrons donc intégrer notre description à RDF.

En utilisant la notation OpenMath, nous avons alors une description du premier service de la forme :

```
<rdf:Description rdf about="serv1">
  <p:parameters>
    <rdf:Seq>
     <rdf:li nom="A" type="Matrix" />
     <rdf:li nom="B" type="Matrix" />
    </rdf:Seq>
  </p:parameters>
  <r:result>Matrix></r:result>
  <n:nom>addition</n:nom>
  <s:specification rdf:parsetype="Literal">
    <OMA>
      <OMS cd="arith1" name="plus"/>
      <OMV name="a"/>
      <OMV name="b"/>
    </OMA>
  </s:specification>
</rdf:Description>
```

[13] `http://www.w3.org/Math/`

[14] `http://www.openmath.org/`

1.2.2.3 Technique de description : UDDI

UDDI[15] (Universal Description, Discovery and Integration) propose un service de pages blanches, c'est-à-dire une recherche par nom, un service de pages jaunes, c'est-à-dire une recherche par classification et catégorisation des services. Il inclut aussi un service de pages vertes, qui définissent les informations techniques pour communiquer avec un web service particulier.

Nous retrouvons une approche similaire aux deux autres précédemment exposées.

1.2.2.4 Les limites de l'approche

Les inconvénients de cette approche sont liés au langage naturel, qui introduit toujours des ambiguïtés. Il n'est pas facile de deviner quel mot-clé sera utilisé et même quelle langue sera utilisée. Pour que cette technique donne de bons résultats, il faudrait donc être sûr que la personne qui décrit le service et la personne qui le recherche utilisent « le même langage », sinon ils pourront utiliser des termes différents pour un même concept ou utiliser un même terme pour représenter deux concepts différents.

De plus, même s'il paraît assez simple de s'accorder sur une langue et éventuellement sur quelques mots-clés, il reste difficile à partir de ces mots-clés de décrire un service complexe qui compose plusieurs opérateurs du domaine. Ceci est en effet très courant dans les bibliothèques d'algèbre linéaire qui proposent en fait des compositions de plusieurs services simples pour améliorer le partage des données et les performances. Nous remarquons que cet exercice est déjà non trivial pour le service 3.

Une première « amélioration », pour aller dans ce sens, peut être trouvée dans RDFS (pour « RDF Schema »), qui offre la possibilité de définir un modèle de méta-données afin de donner du sens aux propriétés associées à une ressource et formuler des contraintes sur les valeurs associées à une propriété afin de lui assurer aussi une signification. Les ontologies dans le cadre du Web Semantique permettent de répondre à certains de ces problèmes.

1.2.3 Web sémantique : les ontologies

Les ontologies ont été introduites pour résoudre le problème du manque d'expressivité, de complétude et de cohérence des descriptions précédentes. Elles doivent permettre à l'utilisateur de décrire des domaines explicitement et rigoureusement.

Dans [AvH03], les auteurs citent les cinq principaux objectifs suivants :

- une syntaxe bien définie ;
- une sémantique bien définie ;
- un support de raisonnement efficace ;
- une expressivité suffisante ;
- et une facilité d'expression.

OWL[16] [MvH04] (qui a pris la succession de DAM+OIL[17]) est le standard W3C pour les ontologies. Pour résoudre le problème d'expressivité et de performance du raisonnement sur les ontologies, trois sous-langages différents ont été définis :

- OWL Full
  - est très expressif et totalement compatible avec RDF ;
  - mais est indécidable.

[15] `http://www.uddi.org/`

[16] `http://www.w3.org/2004/OWL/`

[17] `http://www.daml.org/2000/12/reference.html`

- OWL DL (Description Logic)
  - perd la compatibilité complète avec RDF ;
  - mais gagne le raisonnement efficace et la décidablilité .
- OWL Lite
  - a un raisonnement efficace et décidable et est facile à utiliser ;
  - mais est très restreint côté expressivité.

Les ontologies se basent sur la description de classes, de relations entre les classes (héritage, exclusion, ...) et de propriétés sur les classes et les objets instances de classes.

Le raisonnement sur les ontologies OWL est possible car OWL est associée à des logiques de description ce qui permet d'utiliser des outils de déduction tels que FaCT / FaCT++[18] et RACER[19].

Le problème des ontologies est que nous ne maîtrisons pas le mécanisme de déduction sous-jacent. Ainsi, il faut faire un choix entre l'expressivité et la décidabilité du raisonnement.

Notre problème pourrait être décrit grâce à une ontologie. Des outils tels Protégé[20] [NSD+01] qui possède un greffon OWL [KFNM04] rendent ces descriptions relativement accessibles aux utilisateurs novices. Néanmoins, nous ne maîtrisons pas la logique de déduction sous-jacente et les algorithmes qui interviennent pour décider de l'égalité de deux descriptions. Nous savons que, pour être précise, cette description sera complexe et qu'il sera nécessaire d'appliquer des heuristiques particulières de déduction pour améliorer les performances (**R5**). Cela ne sera pas possible avec les ontologies puisque nous sommes tributaire de la logique associée. Pour la modifier, il faudrait coder de nouvelles heuristiques. Nous perdons alors le bénéfice d'utiliser les ontologies pour leur mécanisme de raisonnement associé.

1.2.4 Les spécifications formelles en génie logiciel

Nous venons de voir que des descriptions semi-formelles, telles les signatures ou les mots-clés (sémantique pas clairement définie ou trop pauvre), ne permettent pas de résoudre le problème de façon satisfaisante. Nous avons également vu qu'une description basée sur des ontologies entraîne une indécidabilité qui ne nous convient pas, ainsi qu'une dépendance trop forte à la logique sous-jacente. De plus, au moment où nos travaux ont débuté, les ontologies en étaient à leurs débuts et tous les outils n'étaient pas encore disponibles ou complets. Nous nous sommes donc intéressés aux spécifications formelles en génie logiciel. Notons que les ontologies sont très proches de certaines techniques de spécifications à base de logique des prédicats.

1.2.4.1 Logique des prédicats du premier ordre (langage Prolog)

Prolog[21] est un langage de programmation logique basé sur le calcul des prédicats du premier ordre. Il exploite trois constructions : les faits, les règles (représentées à l'aide de

[18] `http://owl.man.ac.uk/factplusplus/`
[19] `http://www.racer-systems.com/`
[20] `http://protege.stanford.edu/`
[21] `http://www.swi-prolog.org/`, `http://gnu-prolog.inria.fr/`

clauses de Horn) et les requêtes. Les deux premiers éléments forment une base de connaissance sur laquelle il est possible de raisonner par déduction (mécanisme de résolution) pour répondre à une requête.

En décrivant les propriétés de notre domaine d'application grâce à des règles, la comparaison des services réalisables et de la requête de l'utilisateur serait alors une requête à laquelle Prolog devrait répondre.

Prolog possède un mécanisme pour construire toutes les solutions possibles, en utilisant un parcours en profondeur d'abord. Il nous renverra donc toutes les affectations possibles pour les paramètres du service avec lequel est comparé la requête (y compris s'il y en a une infinité).

Néanmoins, les obstacles à l'utilisation de Prolog sont :

- la représentation des règles sous forme de clauses de Horn. En effet ces règles sont orientées. Dans notre représentation du domaine ce sont des égalités que nous voulons spécifier. Nous serons donc amenés à introduire une clause pour chaque sens d'orientation de notre équation. Cela conduira à la possibilité pour Prolog de boucler.
- l'ordre dans lequel est parcouru l'arbre de recherche. En effet, Prolog réalise un parcours en profondeur d'abord. Il pourra donc être amené à boucler sur des solutions infinies ou sur une répétition de solutions (s'il boucle sur deux clauses représentant les deux orientations d'une même équation) sans trouver toutes les solutions finies au préalable. Il n'est donc pas directement semi-décidable, un effort de codage particulier est nécessaire pour qu'il trouve les solutions finies en premier.
- la difficulté d'appliquer des heuristiques particulières pour l'ordre d'application des règles. En effet, pour résoudre les problèmes évoqués ci-dessus, il faudrait mettre en place des mécanismes pour changer la stratégie de construction des solutions. Cela consiste à programmer un algorithme particulier en utilisant Prolog comme un langage quelconque sans bien exploiter ses spécificités.
  Ces heuristiques particulières peuvent être également nécessaires pour améliorer les performances de l'algorithme, comme évoqué dans les perspectives.

L'utilisation de Prolog, nous aurait permis de reprendre le mécanisme de déduction mais aurait entraîné une difficulté dans la mise en place d'heuristiques pour l'ordre d'application des règles. En effet, Prolog est intéressant si nous souhaitons exploiter son parcours en profondeur d'abord. Il est facile de borner cette profondeur, mais dès que nous voulons modifier ce parcours, ajouter des contraintes sur l'ordre d'application des règles ou de parcours des branches, Prolog n'est alors plus adapté.

Voici un exemple très élémentaire de ce qui peut être réalisé avec Prolog. Nous bornons la profondeur de recherche pour que Prolog ne boucle pas sur une banche infinie. Nous décrivons les propriétés d'associativité et de commutativité de l'addition, ainsi que son élément neutre. Nous obtenons alors le programme Prolog suivant :

```
equal(X,X,P):- P > 0.

equal(S,R,P) :-                    /*x+y=y+x*/
  P > 0,
  PP is P - 1,
  equal(S,add(X,Y),PP),
  equal(add(Y,X),R,PP).
```

```
equal(S,R,P) :-                 /*x->0+x*/
  P > 0,
  PP is P - 1,
  equal(S,X,PP),
  equal(add(zero,X),R,PP).

equal(S,R,P) :-                 /*0+x->x*/
  P > 0,
  PP is P - 1,
  equal(S,add(zero,X),PP),
  equal(X,R,PP).

equal(S,R,P) :-                 /*x+(y+z)->(x+y)+z*/
  P > 0,
  PP is P - 1,
  equal(S,add(X,add(Y,Z)),PP),
  equal(add(add(X,Y),Z),R,PP).

equal(S,R,P) :-                 /*(x+y)+z->x+(y+z)*/
  P > 0,
  PP is P - 1,
  equal(S,add(add(X,Y),Z),PP),
  equal(add(X,add(Y,Z)),R,PP).
```

Nous voyons ici que deux clauses sont nécessaires pour une même équation, comme cela a été évoqué précédemment.

Si nous posons la requête (comparaison de x+y et a+b) :

```
?- findall((X,Y),equal(add(X,Y),add(a,b),3),R).
```

Prolog va renvoyer la réponse suivante :

```
R = [ (a, b),
      (b, a),
      (a, b),
      (add(a, b), zero),
      (a, b),
      (b, a),
      (zero, add(a, b)),
      (zero, add(..., ...)),
      (..., ...)|...]
```

Cela répond à notre problème. Néanmoins, nous avons choisi de ne pas utiliser Prolog pour pouvoir, dans la suite, utiliser des heuristiques pour améliorer les performances de l'algorithme (**R5**). Sur cet exemple très simple, avec seulement trois équations (et cinq clauses), avec la valeur 4 pour P, Prolog met un peu plus de quatre secondes pour répondre (calculé avec le commande *time* de swi-prolog), mais si nous choisissons la valeur 5, après plusieurs heures les réponses ne sont toujours pas obtenues et les capacités CPU et mémoire de l'ordinateur utilisé ne suffisent plus. Bien sûr c'est une version rudimentaire et beaucoup

d'optimisations peuvent être apportées, mais cela donne une idée des problèmes de performances que nous aurions rencontrés et de ce que nous aurions dû mettre en place pour les résoudre. Nous conjecturons qu'en fait nous aurions été amenés à coder en Prolog l'équivalent de nos algorithmes codés en OCaml avec plus de complexité dans l'écriture et moins de possibilités de passage à l'échelle.

1.2.4.2 Assistants de preuves et logiques d'ordre supérieur

Les assistants de preuve permettent d'aider à prouver des théorèmes mathématiques ou des assertions relatives à l'exécution de programmes informatiques, en s'appuyant sur l'isomorphisme de Curry Howard et le calcul des constructions ou la logique d'ordre supérieur. Plus exactement, ils permettent de vérifier qu'une preuve construite par l'utilisateur est correcte en s'appuyant sur les règles de déduction du système formel et sur des tactiques de décision pour certains sous-ensembles du système. Parmi ces assistants de preuve, nous pouvons citer HOL[22] [Gor88], PVS[23] [ORSSC98], Isabelle[24] [NPW02] et Coq[25] [The04].

En fait, la preuve n'est pas construite auparavant puis validée. Mais l'utilisateur ne peut construire qu'une preuve correcte. Pour cela, l'utilisateur décompose l'objectif en sous-objectifs à l'aide des règles de déduction internes (qu'il est possible d'enrichir en en définissant de nouvelles). L'assistant peut résoudre automatiquement certains objectifs (tactiques), mais ne peut pas tous les résoudre (conséquence de l'incomplétude de la plupart des théories). Pour ces derniers, l'utilisateur doit intervenir « manuellement » pour résoudre les sous-buts afin d'aboutir à un problème que l'assistant sait résoudre par une tactique.

Ces assistants de preuve ne répondent donc pas totalement à notre problème, mais nous pourrions les utiliser en définissant des ensembles avec liens de compatibilité pour les types, de nouvelles variables pour les opérateurs et constantes (éléments qui décrivent le domaine d'application), de nouvelles hypothèses pour les équations et pour les services. Et la requête est alors la preuve que nous voulons réaliser.

Illustrons le principe sur un exemple simple : un type de donnée $Matrix$, un opérateur $plus$ (+), une constante O et deux données de l'utilisateur a et b. Nous ajoutons la propriété de la commutativité de $plus$ et O comme élément neutre de $plus$. Nous supposons qu'il y a deux services réalisables :

- $x + y$
- $x + (y + z)$

La requête de l'utilisateur est $a + b$.

Le fichier suivant est une façon simple d'exprimer le problème en Coq, où la primitive $realise$ exprime le fait que nous savons réaliser le service. Cette version est élémentaire et ne prend en compte qu'un unique type, elle doit bien sûr être enrichie dans le cas général.

```
Section Exemple.
Variable Matrix : Set.
Variable plus : Matrix -> Matrix -> Matrix.
Variable O : Matrix.
Variable a : Matrix.
Variable b : Matrix.
Variable realise : Matrix -> Prop.
```

[22] http://hol.sourceforge.net/
[23] http://pvs.csl.sri.com/
[24] http://isabelle.in.tum.de/
[25] http://coq.inria.fr/coq-fra.html

```
Hypothesis plus_comm :
  forall x y:Matrix,
  plus x y = plus y x.                      /*x+y=y+x*/
Hypothesis plus_assos :
  forall x y z:Matrix,
  plus x ( plus y z) = plus( plus x y ) z.  /*x+(y+z)=(x+y)+z*/
Hypothesis O_neutre_plus :
  forall x :Matrix,
  plus x O = x.                             /*x+0=x*/

Hypothesis serv2 :
  forall x y z:Matrix,
  realise ( plus x ( plus y z) ).           /*x+(y+z)*/
Hypothesis serv1 :
  forall x y:Matrix,
  realise ( plus x y ).                     /*x+y*/

Lemma req: realise ( plus a b).             /*a+b*/

auto.
Qed.

End Exemple.
Check req.
```

L'assistant Coq, répond :

```
forall (Matrix : Set) (plus : Matrix -> Matrix -> Matrix)
  (a b : Matrix) (realise : Matrix -> Prop),
(forall x y : Matrix, realise (plus x y)) -> realise (plus a b)
```

Si nous supprimons le premier service, Coq ne sait pas répondre sans l'assistance de l'utilisateur.

Le mécanisme de raisonnement associé aux assistants de preuve ne nous convient donc pas. En effet, la nécessité d'une éventuelle intervention lors du traitement de la requête est contraire à un de nos besoins (**R3**). Cette intervention nécessite une connaissance de l'assistant de preuve et de la logique sous-jacente. De plus, les assistants indiquent si le théorème est vrai ou faux et peuvent renvoyer une trace de l'exécution. Néanmoins, pour connaître tous les services répondant à la requête, il faudrait pouvoir engendrer toutes les preuves possibles et être capable de reconstruire le service ou la composition de services à partir de la trace de l'assistant de preuve. Certains outils permettent de synthétiser le programme correspondant à la preuve ce qui lève notre seconde objection. La première reste valide : les tactiques s'arrêtent à la première solution trouvée (une seule preuve correcte suffit). Si nous souhaitons qu'elles calculent toutes les solutions possibles, il faudrait reprendre partiellement celles-ci et perdre ainsi le bénéfice d'utiliser un outil existant.

Le projet FoCal [26] [HR04, PD02] a pour but de construire un environnement pour développer des bibliothèques certifiées. Cet environnement permet de développer une application en partant d'une spécification et en générant une implantation en OCaml. Grâce à

[26] `http://focal.inria.fr/site/index.php`

une traduction du langage de spécification FoCal vers l'assistant de preuve Coq, il est ainsi possible de prouver que l'implantation est conforme à la spécification donnée.

FoCal est basé sur la notion d'*espèces* qui peuvent être vues comme une liste de méthodes et se rapproche de la notion de classes des langages à objets. Chaque méthode peut être entièrement définie ou être simplement déclarée. Il est possible de créer des liens d'héritages entre les différentes espèces. L'héritage multiple est accepté.

La partie de ce projet qui nous intéresse est la partie spécification des services. En effet, nous devons choisir une méthode de description pour les services. Dans ce projet, les descriptions se rapprochent des langages à objets, nous retrouvons les interfaces et les méthodes des objets. Cette représentation n'est pas la plus adaptée à notre contexte, car nous devons être en mesure de décrire toutes les bibliothèques. L'avantage de ce projet était la traduction du langage FoCal en Coq pour permettre d'avoir un raisonnement sur cette description. Mais comme nous venons de le voir, Coq ne permet pas de répondre à tous nos besoins. FoCal ne le permet pas non plus.

1.2.4.3 Logique de Hoare

En logique de Hoare [Hoa69, Hoa83], les programmes sont vus comme des relations entre deux prédicats : une pré-condition et une post-condition. Si le programme démarre dans un état qui satisfait la pré-condition et qu'il termine, alors l'état final satisfera la post-condition. Nous avons alors une correction totale. Si nous nous contentons d'une correction partielle la terminaison n'est pas exigée.

Les expressions de la logique de Hoare sont de la forme $\{\phi\}S\{\psi\}$, où ϕ et ψ sont des propriétés exprimées dans la logique des prédicats. Cette expression signifie que si les variables de S satisfont la formule ϕ, alors, après exécution de S et si S termine, ces variables satisferont ψ.

Pour pouvoir raisonner sur un programme, il faut connaître les propriétés des opérations élémentaires qu'il invoque. La définition de cet ensemble d'axiomes est très importante. Il faut également trouver une façon explicite de représenter ces post- et pré-conditions pour qu'elles décrivent ce que le programme réalise.

Nos exemples pourraient être représentés de la façon suivante :

- {A :matrix $\wedge$ B :matrix $\wedge$ taille(A)=taille(B) $\wedge$ retour :matrix}
  retour :=serv1(A,B)
  {retour=A+B $\wedge$ taille(retour)=taille(A)}
- {A :matrix $\wedge$ B :matrix $\wedge$ taille(A)= m $\times$ n $\wedge$ taille(B)= n $\times$ p $\wedge$ retour :matrix}
  retour :=serv2(A,B)
  {retour=A*B $\wedge$ taille(retour)= m $\times$ p}
- {A :matrix $\wedge$ B :matrix $\wedge$ C :matrix $\wedge$ D :matrix $\wedge$ retour :matrix
  $\wedge$ taille(A)= m $\times$ n $\wedge$ taille(B)= n $\times$ p $\wedge$ taille(C)= m $\times$ r $\wedge$ taille(D)= r $\times$ p }
  retour :=serv3(A,B,C,D)
  {retour=A*B+C*D $\wedge$ taille(retour)= m $\times$ p}
- {A :real $\wedge$ B :matrix $\wedge$ retour :matrix}
  retour :=serv4(A,B)
  {retour=A*B $\wedge$ taille(retour)= taille(B)}

Nous voyons apparaître ici la nécessité de pouvoir exprimer des propriétés sur les variables et sur les opérations réalisées sur ces variables. La logique de Hoare à elle seule ne permet pas de décrire un service mais offre un support de raisonnement sur les descriptions.

1.2.4.4 Description ensembliste : Méthodes Z et B, VDM

Nous venons de voir que la logique de Hoare à elle seule ne suffit pas pour décrire les services, il faut un langage pour exprimer les pré- et post-conditions. Une famille de solutions consiste à utiliser la théorie des ensembles telles la méthode B [Abr96], le langage Z[27] ou VDM[28] (The Vienna Development Method) [FLM+05].

En s'appuyant sur la notation B et sur la logique de Hoare, nous pouvons décrire le premier service de la façon suivante (en supposant que les matrices sont des matrices carrées) :

$\{\exists N : \quad A \in 1...N \rightarrow 1..N \rightarrow \mathbb{R}$
$\qquad\quad B \in 1...N \rightarrow 1..N \rightarrow \mathbb{R}$
$\qquad\quad retour \in 1...N \rightarrow 1..N \rightarrow \mathbb{R}\}$
retour :=serv1(A,B)
$\{\forall i, j \in 1...N, retour[i, j] = A[i, j] + B[i, j]\}$

Ces notations ne sont pas adaptées à notre problème car elles sont trop précises par rapport à ce que nous souhaitons décrire. De plus, elles ne permettent pas de représenter facilement les équations. Les outils associés à ces méthodes sont des outils de génération de code et ne peuvent donc pas être réutilisés dans notre contexte. Les outils proposés permettent de construire des programmes corrects par un raffinement successif des spécifications en prouvant que chaque raffinement préserve les propriétés (calcul de plus faible pré-condition et preuve semi-automatique de conséquence). Nous retrouvons le même problème que dans le cadre des assistants de preuve et du calcul des prédicats. Les algorithmes partiels de décision ne permettent pas de répondre de manière satisfaisante à nos besoins.

1.2.4.5 Les spécifications algébriques

Les spécifications algébriques sont basées sur la description d'opérateurs et la construction de termes à partir de ces opérateurs. Ces opérateurs et ces termes peuvent être typés ou non. À partir de deux termes, il est possible de définir des équations qui permettent de définir les propriétés des opérateurs.

Exemple 1.2.1
Définition d'opérateurs :
- *+ :* $Int \times Int \rightarrow Int$
- ** :* $Int \times Int \rightarrow Int$

Par exemple, un terme sera alors $+(a, *(b, c))$*. Pour que les termes soient plus facilement lisibles, il est possible d'utiliser une notation infixe :* $a + (b * c)$.

Nous voyons apparaître ici une facilité de description des fonctionnalités des services basée sur les opérateurs principaux du domaine. La présence d'équations permet de décrire des propriétés (commutativité, distributivité, associativité, élément neutre, élément absorbant, ...) et ainsi de raisonner sur la fonctionnalité réalisée.

Les paramètres apparaissant explicitement dans la description, il sera possible de connaître les valeurs auxquelles ils doivent être assignés pour répondre au problème de l'utilisateur.

[27] `http://www.afm.sbu.ac.uk/z/`
[28] `http://www.csr.ncl.ac.uk/vdm/`

En fait, cette notation inspirée de l'algèbre générale est naturellement bien adaptée à la description des mathématiques (MathML et OpenMath sont des formes XML des structures algébriques des termes) et la forme des descriptions répond donc bien à nos objectifs.

Il existe deux familles de méthodes de comparaison de deux termes d'une algèbre : celle basée sur la théorie de la réécriture et celle basée sur l'unification équationnelle et ses dérivées (unification, filtrage, filtrage équationnel, avec ou sans type, ...). Les méthodes de la première famille nécessitent une transformation de l'ensemble des équations en un système de réécriture en s'appuyant sur une relation d'ordre bien fondée sur les termes. Elles sont plus performantes que celles de la seconde famille, mais celles-ci ne demandent pas d'informations complémentaires.

Nous reviendrons plus en détail sur ces spécifications et ces méthodes de comparaison dans le chapitre suivant.

1.2.4.6 Bilan sur les méthodes formelles

Nous avons vu que dans le cas des méthodes formelles nous nous ramenions souvent à exprimer des termes sur des algèbres, avec différentes méthodes de comparaison de ces termes.

Dans le cas de Prolog, la description est intéressante même si elle pose quelques problèmes vis à vis des équations. Cependant le mécanisme de raisonnement associé et le parcours de l'arbre de recherche manquent de flexibilité pour être adaptés à nos besoins. Actuellement, le second algorithme proposé utilise les mêmes mécanismes que Prolog, mais certaines optimisations auraient été difficiles à coder, tout comme certaines des améliorations évoquées en perspectives.

Dans le cas des assistants de preuve, le problème vient de l'arrêt à la première solution et du besoin de l'intervention extérieure de l'utilisateur pour résoudre certains problèmes.

La logique de Hoare est intéressante mais très riche et complexe. De plus, nous avons besoin d'un langage pour décrire les pré- et post-conditions. Nous pouvons choisir de les exprimer par des termes sur une algèbre, comme cela a été fait dans l'exemple. Cela entraîne une comparaison complexe. En optant pour une description basée sur les spécifications algébriques et en ne considérant que les types en entrée et sortie, ainsi que la fonctionnalité, nous représentons un sous-ensemble de la logique de Hoare. Nous discuterons dans les perspectives de la façon dont nous pouvons compléter la description utilisée pour nos travaux afin d'accepter des propriétés aussi complexes que celles proposées ici.

Les descriptions ensemblistes sont quant à elles trop précises pour ce que nous souhaitons faire.

Dans toutes les méthodes existantes, nous voyons apparaître un problème de réutilisation. Les mécanismes ne s'adaptant pas exactement à nos besoins, nous serions obligés de les modifier, le travail serait alors considérable et nous perdrions le bénéfice de l'outil existant.

Nous avons donc choisi une description à l'aide de spécifications algébriques et nous avons choisi de définir notre propre méthode de comparaison pour pouvoir avoir une maîtrise totale des raisonnements appliqués sur les descriptions. Cela permettra d'appliquer des heuristiques particulières pour optimiser l'algorithme en tenant compte de nos contraintes particulières.

1.2.5 Synthèse

Rappelons les besoins exprimés :

R1 Les procédures de courtage (comparaison de la requête et des services) doivent être semi-décidables.

R2 La description d'un domaine d'application doit être réalisable par un spécialiste de ce domaine qui n'est pas spécialiste du formalisme choisi.

R3 Le traitement de cette description doit être complètement automatisé.

R4 La description doit être assez précise pour que la comparaison puisse permettre l'extraction des valeurs des paramètres des services.

R5 La méthode de comparaison doit répondre à des contraintes de performance, nous devons donc pouvoir implanter des heuristiques d'accélération de l'algorithme.

Description \ Contraintes	**R1**	**R2**	**R3**	**R4**	**R5**
Signatures	X	X	X		X
Mots-clés	X	X	X		X
Ontologies	?[a]	X[b]	X	X	
Logique des prédicats	X[c]	X	X	X	
Logique d'ordre supérieur	X	X		X	
Description ensembliste et logique de Hoare	X	X	X	X	X
Spécifications algébriques et réécriture	X	X		X	
Spécifications algébriques et unification	X	X	X	X	X

[a] Dépend du langage de description choisi

[b] Avec l'aide d'outils adaptés

[c] En bornant la profondeur de recherche dans Prolog

Nous voyons clairement que les spécifications formelles sont celles qui répondent le mieux à nos contraintes. Notre choix se portera donc sur l'une d'entre elles. Les descriptions ensemblistes associées à la logique de Hoare étant très complexes et Prolog n'étant pas adapté pour les raisons que nous avons évoqués précédemment, nous avons choisi les spécifications algébriques.

1.3 Les projets existants

Nous allons maintenant exposer plusieurs projets avec lesquels nous avons des préoccupations communes de description et/ou de recherche de services. Ils utilisent tous des spécifications formelles ou des ontologies. Certains se placent dans des domaines spécifiques, d'autres sont génériques.

1.3.1 Domaines applicatifs spécifiques

Nous nous sommes particulièrement intéressés à cinq projets. Ils se situent majoritairement autour de problématiques issues du milieu des mathématiques (FLAME, Falcon, Mizar et Monet). Le cinquième, Spiral, se situe dans le domaine du traitement du signal.

1.3.1.1 FLAME

Le projet FLAME[29] [GGHv01, GvH00] (Formal Linear Algebra Method Environment), comme son nom l'indique, se place dans le cadre de l'algèbre linéaire.

L'objectif du projet FLAME est de faciliter le développement de bibliothèques d'algèbre linéaire dense, pour que ce développement ne soit plus réservé aux experts, mais qu'il soit aussi accessible aux novices. Le projet propose une nouvelle notation pour exprimer les algorithmes, une méthodologie pour une dérivation systématique des algorithmes, des APIs pour représenter les algorithmes ainsi que des outils pour une dérivation mécanique des implantations et des analyses des algorithmes et de leurs implantations.

Ces mécanismes de transformations simplifient la preuve de la correction de l'implantation.

Nous nous sommes intéressés à ce projet pour voir comment sont représentées les données et les algorithmes. Ces descriptions sont beaucoup trop précises pour que nous puissions les utiliser. En effet, une description trop précise entraîne une comparaison beaucoup trop coûteuse. De plus, nous ne sommes pas intéressés par le fonctionnement interne de l'algorithme mais par le service externe rendu.

Dans FLAME, nous retrouvons le souci de rendre accessible à tous ce qui est actuellement réservé à des spécialistes. L'interface proposée est donc composée de fonctions avec des noms explicites. Nous pourrions nous servir de ces noms de méthodes pour représenter les opérations élémentaires de l'algèbre linéaire. Néanmoins, une approche uniquement basée sur le nom des méthodes n'est pas envisageable car chaque bibliothèque possède ses propres conventions qui ne sont pas généralisables.

1.3.1.2 Falcon

Falcon[30] (Fast Array Language COmputatioN) [DGG+94] est un environnement pour un prototypage rapide pour le développement et l'utilisation de programmes numériques « haute performance » et de bibliothèques de calcul scientifique. Il est développé au « Center for Supercomputing Research and Development » de l'université de l'Illinois à Urbana-Champaign. Cet environnement combine les techniques de compilation avec celles utilisées par les développeurs. Il utilise MATLAB comme langage source et engendre soit du Fortran 90 et des directives pour la parallélisation soit du pC++.

Ce projet se limite à MATLAB en entrée (langage orienté vecteur) et se situe lui aussi à un niveau plus précis que ce que nous souhaitons. Notre but n'est pas de transformer du code mais de rechercher dans les bibliothèques existantes uniquement à partir d'une description, ce qui permet de la résoudre.

Notons que les deux projets que nous venons de présenter seraient des bonnes bases pour rechercher des implantations particulières mais cela dépasse l'objectif de nos travaux.

1.3.1.3 Mizar

Le projet Mizar[31] est un projet qui se place également dans le monde des mathématiques. Son but, à long terme, est de développer un logiciel pour aider les mathématiciens

[29] `http://www.cs.utexas.edu/users/flame/`

[30] http ://www.csrd.uiuc.edu/falcon/falcon.html

[31] `http://mizar.org/`

dans l'écriture de leurs articles. Le but est de vérifier la logique mathématique des articles en allant jusqu'à suivre les références à d'autres articles. Pour cela, il faut que tous les articles soient écrits dans le même formalisme et accessibles (stockés dans la même base de données).

Dans chaque article, il est défini du « vocabulaire », qui représente l'ensemble des termes utilisés par la suite pour décrire les théorèmes. Des signatures sont définies pour expliquer comment le vocabulaire peut être utilisé. Tous les constructeurs peuvent être surchargés. Nous retrouvons ici une description à l'aide de spécification algébrique. Ces éléments sont ensuite utilisés, accompagnés de formules logiques, pour définir les théorèmes énoncés dans l'article. Ils peuvent alors être prouvés.

Voici un exemple issu des premiers articles écrits au format de Mizar :

```
------------------------ HIDDEN ----------------------------
:: Built-in Concepts
::  by Andrzej Trybulec

definition                    :: définition des ensembles
  mode set;
end;

definition let x,y be set;
  pred x = y;                 :: l'égalité de deux ensembles
   reflexivity;               :: est réflexif et symétrique
   symmetry;
end;

notation
  antonym x <> y for x = y;
end;

definition let x,X be set;
  pred x in X;                :: l'inclusion dans un ensemble
   asymmetry;                 :: est asymétrique
end;
------------------------ TARSKI ----------------------------
:: Tarski Grothendieck Set Theory
::  by Andrzej Trybulec

 reserve x,y,z,u,N,M,X,Y,Z for set;

:: singletons & unordered pairs

definition let y; func { y } means
:: TARSKI:def 1
  x in it iff x = y;              :: définition des singletons
  let z; func { y, z } means
:: TARSKI:def 2
   x in it iff x = y or x = z;    :: définition des paires
  commutativity;
end;

definition let X,Y;
  pred X c= Y means
```

```
:: TARSKI:def 3
   x in X implies x in Y;          :: inclusion d'ensembles
  reflexivity;
end;

definition let X;
   func union X means              :: union d'ensembles
:: TARSKI:def 4
    x in it iff ex Y st x in Y & Y in X;
end;

:: ordered pairs

 definition let x,y;
  func [x,y] equals
:: TARSKI:def 5                    :: une paire ordonnée est
   { { x,y }, { x } };             :: une paire non ordonnée et
 end;                              :: un premier élément
```

Nous retrouvons une description proche des spécifications algébriques, avec la définition du vocabulaire qui inclut les signatures et les propriétés des opérateurs. Les théorèmes sont ensuite exprimés comme des termes sur ce vocabulaire, ce qui permet de faire des preuves de ces théorèmes.

Dans ces travaux, la représentation est intéressante mais le mécanisme de preuve est basé sur l'utilisation d'un démonstrateur automatique de théorèmes et ne répond donc pas à notre problème. En effet, comme dans le cas d'OWL, de Prolog et des assistants de preuve, l'utilisation d'un démonstrateur de théorème demanderait un lourd travail d'adaptation et poserait problème pour la mise en place d'heuristiques.

1.3.1.4 Spiral

Le projet Spiral se place dans le domaine du « traitement du signal digital ». L'objectif de ce projet est de transformer du code pour l'optimiser pour une plate-forme donnée. Pour cela Spiral utilise la structure mathématique particulière du domaine pour transformer et optimiser le code. Spiral permet d'engendrer différents algorithmes.

Le passage de la requête de l'utilisateur au code se fait en deux étapes : une première étape au niveau algorithmique et une seconde au niveau implantation. Dans un premier temps la requête est transformée en formules qui sont optimisées. Puis ces formules SPL (Signal Processing Language) sont transformées en code C ou Fortran qui est lui-même optimisé. Ceci est ensuite suivi d'une phase d'évaluation qui permet de choisir parmi les différentes solutions proposées et grâce à une fonction d'évaluation des coûts, la solution la moins coûteuse.

Le langage SPL est composé de matrices génériques (diag, ...), de matrices particulières (I, ...), de transformations (Fourier, ...) et de constructeurs de matrices (produit, somme directe, ...). Les principaux constructeurs du langage SPL sont contenus dans cette grammaire simplifiée :

$$
\begin{array}{rcll}
\langle \text{spl} \rangle & ::= & \langle \text{generic} \rangle \mid \langle \text{symbol} \rangle \mid \langle \text{transforme} \rangle \mid & \\
 & & \langle \text{spl} \rangle ... \langle \text{spl} \rangle \mid & \text{(produit)} \\
 & & \langle \text{spl} \rangle \oplus ... \oplus \langle \text{spl} \rangle \mid & \text{(somme directe)} \\
 & & ... & \\
\langle \text{generic} \rangle & ::= & \text{diag}(a_0,...,a_{n-1}) | ... & \\
\langle \text{symbol} \rangle & ::= & I_n \mid J_n \mid L^n_n \mid R_\alpha \mid F_2 \mid ... & \\
\langle \text{transform} \rangle & ::= & \mathbf{DFT}_n \mid \mathbf{WHT}_n \mid \mathbf{DCT\text{-}2}_n \mid \mathbf{Filt}_n(h[z]) \mid ... &
\end{array}
$$

Le langage SPL peut être vu comme le langage des termes d'une spécification algébrique particulière dérivée de ses constructeurs.

La première étape qui nous intéresse tout particulièrement, est réalisée en utilisant des « breakdown rules » et des « manipulation rules ». Les premières sont utilisées pour transformer le problème récursivement et les secondes pour optimiser l'algorithme en remplaçant des formules par d'autres moins coûteuses. Ces deux types de règles sont exprimés en SPL. Les « breakdown rules » sont des décompositions des transformations en un produit de matrices creuses qui peut contenir des transformations généralement plus petites. En fait, une « breakdown rule » est une équation où la partie gauche est une transformation et la partie droite une formule SPL. Elles sont notées « $\rightarrow$ » et non « = ». Les « manipulation rules » sont des équations sur deux formules SPL qui ne contiennent aucune transformation. Spiral applique les premières règles de façon à faire disparaître les transformations puis les secondes de façon à optimiser le code sur des opérations de base sur les matrices.
Voici quelques exemples de « breakdown rules » :

- $\mathbf{DFT}_2 \rightarrow F_2$
- $\mathbf{DCT\text{-}4}_2 \rightarrow J_2 R_{13\pi/8}$

et de « manipulation rules » :

- $(L^{mn}_m)^{-1} \rightarrow L^{mn}_n$
- $L^{kmn}_n \rightarrow (L^{kn}_n \otimes I_m)(I_k \otimes L^{mn}_n)$

Dans le cas du projet Spiral, le but recherché n'est pas la correspondance entre une requête et des services. Notre intérêt s'est porté sur la façon dont est représenté l'algorithme à un haut niveau et la façon dont il peut être transformé pour en obtenir un autre. Les « breakdown rules » et les « manipulation rules » sont des règles figées et le mécanisme est un mécanisme ad hoc pour ces règles. Il ne peut donc pas être réutilisé dans notre contexte. Néanmoins, le travail réalisé pour représenter les algorithmes et les propriétés du domaine est très intéressant et pourra être réutilisé dans notre contexte.

Actuellement, le domaine du traitement du signal n'a pas été décrit dans notre formalisme, mais il fait partie des domaines en cours d'étude. Le langage SPL et les règles utilisées dans Spiral servent de base à la représentation des constantes, des opérateurs et des équations. Il joue le rôle d'« expert du domaine ».

1.3.1.5 Monet

Le projet Monet[32] s'inscrit dans le cadre du web sémantique. Le but est de pouvoir offrir des services mathématiques via des web services qui peuvent être accessibles depuis une large variété de logiciels. Le défi est de développer un outil dans lequel de tels services peuvent être décrits avec assez de détails pour permettre à un agent sophistiqué de choisir le

[32] `http://monet.nag.co.uk/cocoon/monet/index.html`

service qui convient à partir d'une analyse des caractéristiques du problème de l'utilisateur. Cet objectif est donc très proche du notre.

Le broker de Monet est le point clé du projet. Il est chargé de résoudre le problème du client. Pour cela, il se base sur des ontologies. En effet, dans le projet Monet, les services sont décrits à l'aide d'ontologies en OWL[33]. En fait, l'ontologie globale se base sur plusieurs sous-ontologies comme la classification GAMS[34], les termes OpenMath[35], les caractéristiques des machines, les problèmes décrits grâce à leurs entrées / sorties et à des pré- et post-conditions, les algorithmes (numérique, symbolique ou les deux), la bibliographie ou les formats d'encodage.

OpenMath OpenMath permet de représenter les entités mathématiques sous forme de documents XML. Les opérateurs sont définis dans des *Content Dictionaries* par une description en langage naturel et une description des propriétés des opérateurs en OpenMath. Nous retrouvons ici aussi une similitude avec les spécifications algébriques, puisque nous définissons des opérateurs et leurs propriétés et que ces opérateurs sont ensuite utilisés pour décrire des termes (objets mathématiques).

Voici un exemple. La description de l'opérateur *plus* commence par une description en langage naturel, puis une formule qui exprime la commutativité d'opérateur « plus » : $\forall a, b,\ plus(a, b) = plus(b, a)$:

```
<CDDefinition>
<Name>plus</Name>
<Description>The symbol representing an n-ary
commutative function plus.</Description>
<OMOBJ>
  <OMBIND>
    <OMS cd="quant1" name="forall"/>
    <OMBVAR>
       <OMV name="a"/>
       <OMV name="b"/>
    </OMBVAR>
    <OMA>
      <OMS cd="relation1" name="eq"/>
      <OMA>
        <OMS cd="arith1" name="plus"/>
        <OMV name="a"/>
        <OMV name="b"/>
      </OMA>
      <OMA>
        <OMS cd="arith1" name="plus"/>
        <OMV name="b"/>
        <OMV name="a"/>
      </OMA>
    </OMA>
  </OMBIND>
```

[33] http://www.w3.org/TR/owl-features/

[34] http://gams.nist.gov/

[35] http://www.openmath.org/cocoon/openmath/index.html

```
</OMOBJ>
</CDDefinition>
```

La formule mathématique $a + b$ s'écrira alors : :

```
<OMA>
  <OMS cd="arith1" name="plus"/>
  <OMV name="a"/>
  <OMV name="b"/>
</OMA>
```

Le raisonnement sur les ontologies est fait grâce à Instance Store[36] et Racer[37].

L'approche de Monet est intéressante mais pose plusieurs problèmes vis-à-vis de nos contraintes de travail. D'abord, un problème est lié à la définition des ontologies : en effet, il est loin d'être trivial de définir des ontologies et nous avions posé comme besoin que l'utilisateur ne devait avoir aucune connaissance particulière en dehors de son domaine (**R2**). Ensuite, Monet est spécialisé dans le domaine des mathématiques. L'approche pourrait être étendue, mais demanderait un travail considérable pour représenter les termes (étendre OpenMath), les classifications (étendre GAMS (Guide to Available Mathematical Software)) et définir les nouvelles ontologies. Pour donner un ordre d'idée, l'ensemble des ontologies du projet Monet nécessite plus de 20 000 lignes.

Nous aurions souhaité comparer notre approche avec celle de Monet, dans le domaine des mathématiques, en exécutant les mêmes exemples dans les deux formalismes. Mais nous n'avons pas eu accès au broker Monet pour des questions de droits. Les sources ont été temporairement disponibles en ligne, mais au moment où nous avons voulu y accéder, les liens étaient des liens « morts ». Après avoir contacté la responsable du site, elle a expliqué que les sources avaient été retirées pour des questions de droits, a demandé pourquoi nous voulions y avoir accès et n'a plus donné suite.

1.3.2 Les projets génériques

Nous allons maintenant parler de deux projets : les Telescoping Language et le projet Amphion qui se placent dans des domaines d'application restreints mais quelconques.

1.3.2.1 Telescoping Languages

Les «Telescoping Languages»[38] [KBC+05] ont été développés pour engendrer des compilateurs haute performance pour des langages de programmation de domaines scientifiques. L'idée est de faire un traitement sur une librairie de composants d'un domaine cible (par exemple une boîte à outils MATLAB), pour produire un compilateur pour ce domaine qui comprend et optimise les briques de base du langage. Il construit une nouvelle librairie en décomposant et recomposant les procédures.

Un des objectifs du projet est d'engendrer du code C ou FORTRAN de haute performance. Pour cela, les composants ne doivent pas être traités comme des boîtes noires. Un des points clés des optimisations réalisées (au niveau des procédures MATLAB ou de S

[36] `http://instancestore.man.ac.uk/`
[37] `http://www.racer-systems.com/`
[38] `http://telescoping.cs.rice.edu/`

(langage d'analyse statistique [BCW88])) est de gagner en précision sur les types réels et les tailles des variables.

Au sein de ce processus, l'outil Palomar joue un rôle particulier. Il prend en entrée un ensemble de procédures exprimées en langage de base qui définissent les primitives du langage du domaine et les composants de la boîte à outils. En sortie, Palomar va engendrer une librairie qui contient les primitives du domaine et des versions particulières des routines définies pour exploiter le contexte dans lequel elles vont être exécutées. La meilleure version du composant sera alors utilisée une fois le contexte connu.

Pour pouvoir optimiser le code, Palomar se base sur des annotations de l'utilisateur au niveau de la librairie. Ces annotations permettent de remplacer des séquences d'invocations de composants par des séquences plus efficaces. Ces annotations peuvent être vues comme définissant une algèbre de relations sur les composants, avec la complexité des services comme base pour les décisions d'optimisation. Les transformations de séquences se font à l'aide de règles qui sont soit engendrées par Palomar, soit fournies par le créateur de la librairie. Ces transformations peuvent être accompagnées de pré- et post-conditions.

Dans l'article [KBC+01], les auteurs présentent un exemple d'axiomes possibles exprimés en langage Z [39]. Ils donnent les exemples de l'associativité, de l'addition de matrices et de la distributivité de la multiplication de matrices par rapport à la multiplication matrice - vecteur.

$\vdash \forall m_1, m_2, m_3 : Matrix \bullet$
$mmadd(m_1, mmadd(m_2, m_3)) = mmadd(mmadd(m_1, m_2), m_3)$
Cet axiome permet de changer l'ordre d'évaluation de l'opération.

$\vdash \forall m_1, m_2 : Matrix;\ v_1 : Vector \bullet$
$mvmul(mmmul(m_1, m_2), v_1) = mvmul(m_1, mvmul(m_2, v_1))$
Cette opération permet de gagner en complexité quand elle est appliquée dans le « bon » sens.

Ces identités sont utilisées de deux façons :

- pour appliquer le meilleur composant dans un contexte donné ;
- pour remplacer des séquences identiques d'appels de fonctions par une autre fonction qui réalise cette séquence de façon optimisée.

Les « Telescoping Languages » présentent une approche intéressante, mais plus dans une optique d'optimisation de code que dans celle de courtage. Le but est de transformer du code en vue de l'optimiser et non de comparer deux spécifications. Les règles qui permettent de transformer le code ne sont appliquées que dans le sens qui permet de réduire le coup d'exécution, donc uniquement des algorithmes moins coûteux sont engendrés.

De plus, seul un prototype très préliminaire du système de génération de compilateur Palomar a été réalisé. Ces travaux ne sont donc pas assez avancés pour pouvoir nous servir de base de travail.

1.3.2.2 Amphion

Ce projet [SWL+94] réalisé au sein de la NASA a pour objectif d'engendrer du code source à partir d'une spécification de haut niveau. Il est indépendant du domaine, mais, comme dans notre cas, s'applique à un domaine particulier. Amphion a été validé sur trois domaines de la NASA : la géométrie du système solaire, la mécanique des fluides et la navigation des navettes spatiales.

[39] `http://vl.zuser.org/`

La partie du projet qui nous intéresse plus particulièrement est le démonstrateur automatique de théorème Snark. Celui-ci sert à prouver que la spécification est un théorème du domaine. Pour cela, il engendre des substitutions obtenues par unification et application d'équations. Snark[40] utilise la réécriture et la règle de para-modulation [R69] pour raisonner sur les égalités du domaine. L'utilisateur doit fournir un ordre strict sur les constantes et les fonctions du domaine. Cet ordre est utilisé pour contrôler la règle de para-modulation. Il est très important et a une grande influence sur le temps d'obtention d'une preuve. Cela implique donc qu'un spécialiste de la réécriture construise cet ordre. Ceci est contraire à une de nos contraintes de travail (**R2**).

Snark prend en compte les types et les sous-types et permet de définir des théories. Ces théories définissent du vocabulaire, des fonctions et des prédicats. Les symboles peuvent être déclarés comme commutatifs et/ou associatifs. Snark intègre un mécanisme de réécriture qui permet de traiter certaines égalités sous forme de règles de réécriture. Il propose aussi une instruction « closure » qui permet d'obtenir plusieurs solutions pour un même problème. Il permet également de raisonner sur des égalités. Cependant, il peut boucler.

Voici deux exemples simples qui montrent pourquoi Snark n'était pas adapté à nos besoins.

```
(load "snark-system.lisp")
(make-snark-system)
(initialize)
(use-resolution t)
(assert '(= (* 1 ?x) ?x ) :name '1-neutre)
(assert '(= (* (* ?x ?y) ?z) (* ?x (* ?y ?z)))
                                    :name '*-assoc)
(prove
  '(= (* (* ?x ?y) ?z) (* a b ))
:answer '(ans ?x ?y ?z))
; rechercher x, y et z tels que ((x*y)*z) = (a*b)
(closure)
```

Snark répond :

```
[...]
; All agendas are empty.
:AGENDA-EMPTY
```

Ce qui signifie qu'il ne peut pas faire la preuve.

Si nous ajoutons l'utilisation de la règle de para-modulation :

```
(use-paramodulation)
```

Snark répond :

```
[...]
(Row 15
   false
   (resolve 4 :code-for-=)
   Answer (ans 1 a b))
```

[40] `http://www.ai.sri.com/snark/tutorial/tutorial.html`

```
)
[...]
:PROOF-FOUND
```

Ce qui signifie qu'il trouve bien la solution et qu'il serait donc nécessaire d'utiliser la règle de para-modulation.

Néanmoins, dans le même contexte mais avec une autre requête :

```
(prove
  '(= (* ?x (+ ?y 1)) (* a b) )
:answer '(ans ?x ?y))
```

Snark va boucler et construire une suite infinie de 1*1*1*1*1*1*1*1*1*1*...

Ce projet est intéressant mais l'approche pose deux problèmes majeurs : l'importance de l'ordre qui nécessite un expert (ce qui viole le besoin **R3**) et le fait que Snark puisse boucler dans certains cas (ce qui viole le besoin **R1**). Nous aurions pu envisager de reprendre les sources de Snark pour les adapter à notre problème, mais la nécessité de l'ordre était une contrainte trop importante.

1.4 Conclusion

Après une étude des différentes descriptions possibles pour les fonctionnalités des services, nous avons choisi de nous orienter vers une description formelle et plus précisément vers une description basée sur les spécifications algébriques.

Après une étude des différents projets existants, nous avons vu qu'aucun ne répondait à tous nos besoins, nous avons donc choisi de développer notre propre mécanisme de comparaison.

Le tableau suivant résume quelques propriétés intéressantes des différents projets présentés.

	Flame	Falcon	Mizar	Spiral	Monet	T.L.	Amphion
Générique						X	X
Description utilisable dans notre contexte			X	X	X		X
Traitement automatique	X	X	X	X	X	X	
Description pouvant être étendue facilement			X	X		X	X
Informations pouvant être intégrées à notre formalisme				X	X		
Description s'appuyant sur la structure de spécification algébrique			X	X			X

Ces deux aspects (description choisie et mécanisme de comparaison) sont développés dans les chapitres suivants.

Nous allons maintenant introduire plus en détail la solution à base de spécification algébrique que nous proposons ainsi que la formulation de l'ensemble des services qui peuvent être rendus dans un domaine donné à partir d'un ensemble existant.

Chapitre 2

Formalisation du problème

Nous avons présenté dans le chapitre précédent différentes techniques de description qui peuvent être utilisées pour exprimer les fonctionnalités rendues par les services et différentes méthodes pour comparer les descriptions. Pour les raisons expliquées précédemment, nous avons choisi de décrire les domaines à l'aide de spécifications algébriques.

Dans ce chapitre, nous allons tout d'abord définir plus précisément les spécifications algébriques et les différentes méthodes de comparaison des termes issus de ces descriptions. Ces méthodes de comparaison sont de deux types : celles basées sur la réécriture et celles basées sur des techniques d'unification équationnelle ou de filtrage équationnel.

Ensuite, nous formaliserons notre problème à l'aide de ces spécifications dans le but de proposer deux algorithmes, dont nous prouverons la correction. L'objectif de ces algorithmes est d'énumérer l'ensemble des solutions qui permettent de rendre le service souhaité par l'utilisateur (appelé requête) à partir de compositions de services existants, de constantes du domaine, et d'applications d'équations du domaine. Notons qu'il s'agit en fait de comparer la requête avec chaque service existant en appliquant les équations et en appelant éventuellement d'autres services. Nos algorithmes doivent donc énumérer les solutions du problème d'égalité entre requête et service. Notre algorithme sera correct si tous les services et compositions de services qu'il retourne permettent effectivement de répondre à la requête. Nous évoquerons également des éléments formels concernant la complétude sans traiter le cas général. Nous utilisons de manière légèrement abusive le terme complétude pour indiquer que notre algorithme peut générer toutes les solutions répondant au problème (éventuellement en temps non fini si ce nombre n'est pas fini). Nous maîtriserons cette non terminaison par une quantité d'énergie allouée par l'utilisateur. Nous aurions également pu utiliser le terme exhaustivité de l'énumération des solutions. Mais le terme complet était plus proche du résultat que nous voulions emprunter à Gallier et Snyder. Cette propriété est moins importante que la correction dans le cadre du courtage. Nous dirons que notre algorithme est complet s'il renvoie tous les services et toutes les compositions de services qui permettent de répondre à la requête de l'utilisateur.

Le premier algorithme est basé sur les travaux de Gallier et Snyder [GS89] sur l'unification équationnelle dans le cas général, avec pour objectif de réutiliser la preuve de la complétude déjà effectuée par les auteurs. Pour des raisons discutées par la suite, nous n'avons finalement pas pu conclure en ce qui concerne cette propriété. Nous présentons quand même cet algorithme car il possède des propriétés intéressantes de séparabilité ; il traite les sous-problèmes engendrés par le problème principal de façon totalement indépendante. Néanmoins, ceci entraîne des problèmes de performance. Un second algorithme a alors été proposé pour pallier ces problèmes de performance. Il réalise un traitement en

séquence des différents sous-problèmes, en propageant les contraintes, ce qui entraîne une détection des erreurs plus rapide et donc moins de calculs effectués. Celui-ci est également plus adapté à des preuves de complétude et nous présenterons celle-ci pour certaines formes d'équations. Ces deux algorithmes sont détaillés dans les chapitres suivants.

Rappelons que notre problème consiste, dans un domaine particulier, à trouver tous les services et compositions de services qui permettent de répondre à une requête. Pour cela, nous nous appuyons sur les connaissances du domaine telles que les propriétés des opérateurs.

Pour réaliser les preuves de correction nous avons besoin de formaliser les services et les compositions de services ainsi que la signification de l'expression « répondre à la requête ». C'est le travail effectué dans ce chapitre. Dans un premier temps, nous allons revenir en détail sur les spécifications algébriques, les techniques de réécriture et les problèmes de la famille de l'unification équationnelle, puis nous allons formaliser nos services et notre requête. Nous développerons ensuite un mécanisme de génération de services et de compositions de services réalisables à partir des services disponibles. Enfin, nous formaliserons ce que signifie « répondre à la requête ».

2.1 Les spécifications algébriques

Dans le chapitre précédent, nous avons présenté un ensemble de descriptions dont une majorité ne convient pas à notre problème à cause d'un manque d'expressivité. Nos contraintes de travail imposent une description précise de la fonctionnalité rendue (**R4**) sur laquelle il est possible de faire un raisonnement en tenant compte des propriétés spécifiques du domaine (**R1**). C'est pourquoi nous nous sommes intéressés aux spécifications algébriques. Ces spécifications permettent de définir les fonctionnalités rendues par un composant logiciel à l'aide des structures de données qu'il manipule. De plus, les notations algébriques sont proches des notations mathématiques. En effet, nous reprochions aux descriptions par mots-clés un problème de signification lié à l'interprétation, à la langue,... Ce problème est en partie levé ici car nous pouvons utiliser les notations standard des mathématiques pour représenter les termes. Une addition sera notée « + », une soustraction « - », ...

2.1.1 Définitions

Les notations et définitions qui suivent sont principalement issues de [Lal90] et [GM92].

2.1.1.1 Signatures et termes

Définition 2.1.1 (S-ensemble)
Soit S un ensemble, un $S - ensemble$ est une famille $A = (A_s)_{s \in S}$ d'ensembles indexée par S.

Définition 2.1.2 (Signature)
Une *signature* est un ensemble Σ muni d'une application $ar : \Sigma \rightarrow \mathbb{N}$. Si $f \in \Sigma$ et $ar(f) = n$, alors f est d'*arité* n ; si $n = 0$, f est une *constante*.

Une *signature hétérogène* est une paire (S,Σ), où :
- S est un ensemble de symboles de *sortes* (appelées aussi *types* dans la suite)
- Σ est un ensemble de symboles représentant des constantes et des fonctions typées

– $c : s$, avec $s \in S$
– $f : s_1 \times ... \times s_n \rightarrow s_{n+1}$, avec $s_1, ..., s_n, s_{n+1} \in S$ et $n \geq 1$

Σ est un $S^* \times S$-ensemble $\{\Sigma_{w,s} | \ w \in S^* \ et \ s \in S\}$.
– $c : s \in \Sigma_{\lambda,s}$
– $f : s_1 \times ... \times s_n \rightarrow s_{n+1} \in \Sigma_{(s_1,...,s_n),s_{n+1}}$

Une *signature hétérogène avec sous-typage* est un triplet (S,$\leq$,Σ), où (S,Σ) est une signature hétérogène, (S,$\leq$) est un ensemble ordonné et les opérateurs satisfont la condition suivante :

$$f \in \Sigma_{w_1,s_1} \cap \Sigma_{w_2,s_2} \ et \ w_1 \leq w_2 \Rightarrow s_1 \leq s_2,$$

où $\leq$ est étendu sur S^* de manière usuelle en utilisant l'ordre lexicographique.
Quand il n'y aura aucune ambiguïté sur l'ensemble S (respectivement sur (S,$\leq$)), nous noterons Σ à la place de (S,Σ) (respectivement de (S,$\leq$,Σ)).

Exemple 2.1.1
Nous pourrons par exemple décrire, de façon très simplifiée, l'algèbre linéaire par cette signature hétérogène :

$S = \{Scalar, Matrix\}$
$\Sigma = \{$
$\quad 0 : Matrix$
$\quad I : Matrix$
$\quad + : Matrix \times Matrix \rightarrow Matrix$
$\quad + : Scalar \times Scalar \rightarrow Scalar$
$\quad * : Matrix \times Matrix \rightarrow Matrix$
$\quad * : Scalar \times Matrix \rightarrow Matrix$
$\quad * : Scalar \times Scalar \rightarrow Scalar$
$\}$

Définition 2.1.3 (Termes sans variable)
Soit Σ une signature, T_Σ est la plus petite partie E de Σ^* telle que
– si $c \in \Sigma$, $ar(c) = 0$, alors $c \in E$;
– si $f \in \Sigma$, $ar(f) = n \geq 1$ et si $M_1, ..., M_n \in E$, alors $f(M_1, ..., M_n) \in E$.
Les éléments de T_Σ sont appelés des *termes*.

Remarque : La notation $f(M_1, ..., M_n)$ plus semblable à un appel de fonction est utilisée à la place de $f \ M_1 \ ... \ M_n$.

Cette définition des termes n'est pas tout à fait classique puisqu'elle ne prend pas en compte les variables.

Définition 2.1.4 (Termes)
Soient Σ une signature et X un ensemble de symboles de *variables*, $T_\Sigma[X]$ est la plus petite partie E de $(\Sigma \cup X)^*$ telle que
– si $c \in \Sigma$, $ar(c) = 0$, alors $c \in E$;
– si $f \in \Sigma$, $ar(f) = n \geq 1$ et si $M_1, ..., M_n \in E$, alors $f(M_1, ..., M_n) \in E$;
– si $x \in X$, alors $x \in E$.
Les éléments de $T_\Sigma[X]$ seront maintenant appelés des *termes*.

Définition 2.1.5 (Variables d'un terme)
Soient Σ une signature, X un ensemble de symboles et $t \in T_\Sigma[X]$, $Var(t)$ est l'ensemble défini par :

- si $c \in \Sigma$, $ar(c) = 0$, alors $Var(c) = \emptyset$;
- si $f \in \Sigma$, $ar(f) = n \geq 1$, alors $Var(f(M_1, ..., M_n)) = \bigcup_{1 \leq i \leq n} Var(M_i)$;
- si $x \in X$, alors $Var(x) = \{x\}$.

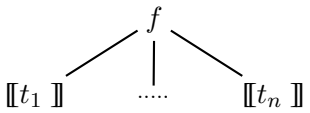
Définition 2.1.6 (Termes typés)
Soient (S,Σ) une signature hétérogène, $X = (X_s)_{s \in S}$ un S-ensemble de variables. Les termes typés sont définis par :

- chaque variable x de X_s est un terme typé de sorte s ;
- chaque constante $c : s$ de Σ est un terme typé de sorte s ;
- si $M_1, ..., M_n$ sont des termes typés de sortes $s_1, ..., s_n$, et $f : s_1 \times ... \times s_n \rightarrow s \in \Sigma$ alors $f(M_1, ..., M_n)$ est un terme typé de sorte s.

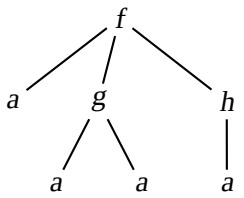
Nous noterons $T_\Sigma[X]_{\ s}$ l'ensemble des termes typés de sorte s.
Posons $T_\Sigma[X] = \bigcup_{s \in S} T_\Sigma[X]_{\ s}$.

Définition 2.1.7 (Termes et arbres)
Soient Σ une signature et X un ensemble de variables, les termes de $T_\Sigma[X]$ peuvent être représentés sous forme d'arbre :

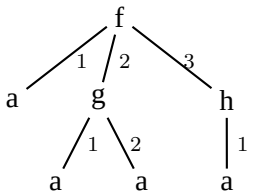
- si $t = c$ est une constante, alors $[\![t\ ]\!]$ est la feuille c ;
- si $t = x$ est une variable, alors $[\![t\ ]\!]$ est la feuille x ;
- si $t = f(t_1, .., t_n)$, alors $[\![t\ ]\!]$ est l'arbre



Exemple 2.1.2
Le terme $f(a, g(a, a), h(a))$ est représenté par l'arbre :



Définition 2.1.8 (Occurrences)
Soient Σ une signature et M un terme représenté par un arbre ; numérotons les arcs issus de chaque nœud de gauche à droite, à partir de 1.



Nous accéderons à un symbole de M à l'aide du mot obtenu en concaténant les numéros des arcs de la branche menant de la racine à ce symbole. Ces mots sont appelés des *occurrences*. Dans la suite, ces occurrences seront aussi appelées *positions*.

Sur l'exemple 2.1.2, ϵ conduit à f, 1 et 21 conduisent à deux a différents et 2 conduit à g.

La donnée d'une occurrence p de M détermine un nœud de l'arbre, le symbole qui l'étiquette et le sous-terme (sous-arbre) qui en est issu.

Pour tout terme M, nous définissons :

- l'ensemble $\mathcal{O}(M)$ des occurrences de M,
- le symbole $M(p)$ en p, pour $p \in \mathcal{O}(M)$,
- le sous-terme $M_{|_p}$ de M en p, pour $p \in \mathcal{O}(M)$

par

- si $M = c \in \Sigma$, alors $\mathcal{O}(M) = \{\epsilon\}$, $M(\epsilon) = c$ et $M_{|_\epsilon} = c$
- si $M = f(M_1, ..., M_n)$, alors
  - $\mathcal{O}(M) = \{\epsilon\} \cup \left(\bigcup_{1 \leq i \leq n} i.\mathcal{O}(M_i) \right)$
  - $M(\epsilon) = f$
  - $M_{|_\epsilon} = M$
  - $M(i.p) = M_i(p)$
  - $M_{|_{i.p}} = M_{i|_p}$

$M_{|_p}$ est un terme.

Définition 2.1.9 (Greffe)
Soient Σ une signature, X un ensemble de variables, M, $N \in T_\Sigma[X]$ et $p \in \mathcal{O}(M)$. Si $m = f(M_1, ..., M_n)$, avec $n \geq 0$, nous posons :

- $M[\epsilon \leftarrow N] = N$
- $M[i.p \leftarrow N] = f(M_1, ..., M_{i-1}, M_i[p \leftarrow N], M_{i+1}, ..., M_n)$

$M[p \leftarrow N]$ est un terme.

2.1.1.2 Substitution

Définition 2.1.10 (Substitution)
Soient Σ une signature et X un ensemble de variables. Toute application $\sigma : X \rightarrow T_\Sigma[X]$, qui est l'identité presque partout (sauf sur une partie finie de X), est appelée *substitution.*

L'ensemble $Dom(\sigma) = \{x \in X |\ \sigma(x) \neq x\}$, qui est donc fini, est appelé *domaine* de σ.

Soit V un ensemble de variables, l'ensemble $etend(\sigma, V) = \bigcup_{x \in V} Var(\sigma(x))$ est l'*étendue* de σ à V.

L'ensemble $Im(\sigma) = etend(\sigma, Dom(\sigma))$ est l'*image* de σ.

Si $Dom(\sigma) = \{x_1, ..., x_n\}$, tous les x_i étant distincts, alors la substitution σ pourra être représentée par $\{x_1 \mapsto \sigma(x_1), ..., x_n \mapsto \sigma(x_n)\}$.

σ est étendue à $T_\Sigma[X]$ par $\widehat{\sigma} : T_\Sigma[X] \rightarrow T_\Sigma[X]$ de la façon suivante :

- $\widehat{\sigma}(c) = c$ pour $c \in \Sigma$ et $ar(c) = 0$
- $\widehat{\sigma}(f(M_1, ..., M_n)) = f(\widehat{\sigma}(M_1), ..., \widehat{\sigma}(M_n))$ si $f \in \Sigma$ et $ar(f) = n$
- $\widehat{\sigma}(x) = \sigma(x)$ si $x \in X$

Dans la suite, l'extension $\widehat{\sigma}$ de σ sera notée σ.

Définition 2.1.11 (Composition)
La *composition* $\delta \circ \sigma$ de deux substitutions est définie par : $\delta \circ \sigma(x) = \sigma(\delta(x))$.

Définition 2.1.12 (Substitution plus générale)
Soient σ et θ deux substitutions. θ est plus générale que σ ($\sigma \geq \theta$) s'il existe une substitution δ telle que $\sigma = \theta \circ \delta$.

Définition 2.1.13 (Idempotente)
Une substitution σ est *idempotente* si et seulement si $\sigma \circ \sigma = \sigma$.

Propriété 2.1.1
Une substitution est idempotente si et seulement si $Dom(\sigma) \cap Im(\sigma) = \emptyset$.

2.1.1.3 Les équations

Définition 2.1.14 (Equation)
Soient Σ une signature, X un ensemble fini de variables et $=$ un symbole relationnel binaire pour l'égalité, une *équation* est de la forme : $\forall x_1 ... x_k \ (M = N)$ où $\{x_1, ..., x_k\} = Var(M) \cup Var(N) = X$ et $M,\ N \in T_\Sigma[X]$. Toutes les équations étant de cette forme, les quantificateurs sont toujours omis.

Dans la suite, $\mathcal{E}$ représentera un ensemble d'équations.

Exemple 2.1.3
Dans le cadre de notre exemple de l'algèbre linéaire simplifiée, $\mathcal{E}$ *contient :*
- $x : Matrix,\ x * I = x$
- $x : Matrix,\ x * O = O$
- $x, y, z : Matrix,\ x * (y * z) = (x * y) * z$

Définition 2.1.15 (Variables libres)
Nous appelons variables libres d'une équation (e_1, e_2), l'ensemble :

$$(Var(e_1) \backslash Var(e_2)) \cup (Var(e_2) \backslash Var(e_1))$$

Elles correspondent aux variables qui ne sont présentes que dans un des deux membres de l'équation.

Définition 2.1.16 (Système d'inférence de la logique équationnelle)
À partir d'un ensemble d'équations $\mathcal{E}$, d'autres équations peuvent être engendrées à partir des règles d'inférences suivantes :

Réflexivité $\dfrac{}{t = t}$

Symétrie $\dfrac{t = t'}{t' = t}$

Transitivité $\dfrac{t = t', t' = t''}{t = t''}$

Remplacement $\dfrac{t_1 = t'_1, ..., t_n = t'_n}{f(t_1, ..., t_n) = f(t'_1, ..., t'_n)}$

Substitution $\dfrac{t = t'}{\sigma(t) = \sigma(t')}$, σ une substitution

2.1.2 Les langages de spécifications algébriques

Cette approche a été mise en œuvre dans de nombreux environnements qui offrent différentes évolutions pour simplifier l'utilisation du formalisme (lisibilité, modularité, réutilisabilité, ...) et ceux-ci proposent les spécifications pour vérifier des propriétés (consistance, complétude, satisfaction, ...), pour engendrer le squelette de programmes ainsi que des tests, pour construire des systèmes de réécriture, pour exécuter la spécification, ...

Nous pouvons citer par exemple ASL [SW83], ASSPEGIC [BC85], LPG [BE86], Larch [GH86], ACT ONE et ACT TWO [CEW93], OBJ3 [GWM$^+$93], ASD+SDF [vDM96], CASL [ABK$^+$02, BM04], ...

Dans le cadre de nos travaux, nous nous sommes limités à un langage élémentaire pour réduire les coûts d'implantation des prototypes que nous avons réalisés. Par la suite, nous envisageons d'utiliser un langage plus riche tel CASL et d'adapter les interfaces WEB et les algorithmes de courtage pour profiter des possibilités offertes.

Nous avons choisi de représenter les domaines par une signature hétérogène avec sous-typage (S,$\leq$,Σ) et un ensemble d'équations $\mathcal{E}$.

Les services sont alors des termes de $T_\Sigma[P]$, où P représente l'ensemble des paramètres du service. La requête est un terme de $T_\Sigma[C]$, où C représente l'ensemble des données de l'utilisateur. Nous reviendrons plus en détail sur la représentation des services et de la requête dans la suite de ce chapitre.

Nous sommes donc amenés à faire une comparaison entre deux termes, modulo un ensemble $\mathcal{E}$ d'équations. Nous allons maintenant aborder différentes méthodes de comparaison de termes.

2.1.3 Comparaison grâce à la réécriture

Dans un premier temps, nous avons naturellement essayé de réutiliser le travail effectué dans le domaine de la réécriture et notamment réutiliser des outils de réécriture existants.

2.1.3.1 La réécriture

Les notations et définitions qui vont suivre sont issues de [BN98].

Définition 2.1.17 (Système de réduction abstrait)
Un *système de réduction abstrait* est une paire (**A**,$\rightarrow$), où la *réduction* $\rightarrow$ est une relation binaire sur **A**. À la place de $(a, b) \in \rightarrow$, nous écrirons $a \rightarrow b$.

Définition 2.1.18 (Fermeture)
La fermeture réflexive, transitive de $\rightarrow$ est notée $\overset{*}{\rightarrow}$, sa fermeture symétrique est notée $\leftrightarrow$ et sa fermeture réflexive, transitive et symétrique est notée $\overset{*}{\leftrightarrow}$.

Définition 2.1.19 (Forme normale)
Nous dirons que :

- x est *réductible* si et seulement si il existe y tel que $x \rightarrow y$.
- x est *une forme normale* si et seulement si x n'est pas réductible.
- y est *une forme normale* de x si et seulement si $x \overset{*}{\rightarrow} y$ et y est une forme normale. Si x a une unique forme normale, elle est désignée par $x\downarrow$.

x et y sont *joignables* si et seulement si $\exists z,\ x \overset{*}{\rightarrow} z \overset{*}{\leftarrow} y$ et nous écrirons $x \downarrow y$.

Définition 2.1.20 (Propriétés d'une réduction)
Une réduction $\rightarrow$ est dite :

- *Church-Rosser* si et seulement si $x \overset{*}{\leftrightarrow} y \ \Rightarrow \ x \downarrow y$.
- *confluente* si et seulement si $y_1 \overset{*}{\leftarrow} x \overset{*}{\rightarrow} y_2 \ \Rightarrow \ y_1 \downarrow y_2$.
- *noéthérienne (qui termine)* si et seulement si il n'y a pas de chaîne infinie $a_0 \rightarrow a1 \rightarrow ....$
- *normalisante* si et seulement si tous les éléments ont une forme normale.
- *convergente* si et seulement si elle est confluente et termine.

Propriété 2.1.2
$\rightarrow$ est Church-Rosser est équivalent à $\rightarrow$ est confluente.

Propriété 2.1.3
Si $\rightarrow$ est confluente et normalisante alors $x \overset{*}{\leftrightarrow} y \ \Leftrightarrow \ x \downarrow = y \downarrow$.

Cette dernière propriété nous donne un résultat très intéressant qui permet de vérifier simplement si deux termes sont en relation, à savoir de vérifier que leurs formes normales sont égales.

Définition 2.1.21 ($\rightarrow_{\mathcal{E}}$)
Soient Σ une signature et $\mathcal{E}$ un ensemble d'équations sur Σ. La relation de réduction $\rightarrow_{\mathcal{E}}$ est définie par :
$s \rightarrow_{\mathcal{E}} t$ si et seulement si
$\exists (l, r) \in \mathcal{E}$, $p \in \mathcal{O}(s)$ et σ une substitution,
tels que $s_{|_p} = \sigma(l)$ et $t = s[p \leftarrow \sigma(r)]$,
où $\mathcal{O}(s)$ désigne l'ensemble des occurrences de s défini précédemment.

Définition 2.1.22 (Règle et système de réécriture)
Une *règle de réécriture* est une équation $l = r$ telle que l n'est pas une variable et $Var(l) \supseteq Var(r)$. Dans ce cas, nous écrirons $l \rightarrow r$ au lieu de $l = r$. Un *système de réécriture* est un ensemble de règles de réécriture.

Théorème 2.1.1
Si R est un système de réécriture fini et convergent, $=_R$ est décidable :

$$s =_R t \Leftrightarrow s \downarrow_R = t \downarrow_R$$

Notre problème est la comparaison de deux termes modulo une théorie équationnelle $\mathcal{E}$. Une solution simple et élégante consiste à transformer $\mathcal{E}$ en un système de réécriture équivalent, fini et convergent (confluent et terminant). En effet, notre problème se ramènerait alors à la comparaison des deux formes normales.

Pour prouver ces propriétés, il faut connaître des informations sur la théorie équationnelle, ce qui n'est pas notre cas, puisqu'une des hypothèses de travail est l'absence de contraintes sur les équations. Néanmoins, il existe des méthodes pour transformer un système $\mathcal{E}$ fini mais quelconque en système de réécriture fini et convergent.

La complétion de Knuth-Bendix [KB70] est un algorithme qui transforme un ensemble fini d'égalités en un système de réécriture fini, terminant et confluent, dont la réduction préserve l'égalité. Les entrées de cet algorithme sont la théorie équationnelle et une relation d'ordre bien fondée qui permet d'orienter les règles. Notons que cet algorithme est semi-décidable. Il ne termine pas toujours et peut échouer.

2.1.3.2 Outils de réécriture considérés

Les trois principaux outils de réécriture étudiés sont Elan [BKK+98], CîME[1] [CMU04] et Maude [CDE+00, CDE+03] car ils proposent des fonctionnalités équivalentes sur les points qui nous intéressent et notamment une complétion de Knuth-Bendix et l'obtention d'une forme normale à partir d'un système fini et convergent. Néanmoins, CîME ne gérant pas les types et Elan n'acceptant pas le sous-typage, Maude a plus particulièrement retenu notre attention.

Maude permet de définir une signature hétérogène avec sous-typage. Certaines opérations sont alors disponibles pour manipuler les termes de l'algèbre associée. La signature est définie à partir des opérations et d'indications de l'utilisateur sur les types et leurs liens d'héritages. La théorie équationnelle est donnée par des équations sur les opérateurs (l'associativité, la commutativité et l'élément neutre ne sont pas déclarés comme les autres équations mais possèdent des labels spéciaux pour un traitement plus efficace). Maude offre aussi la possibilité de définir des équations conditionnelles. Pour une notation plus lisible, il est également possible d'associer des priorités aux opérateurs.

Maude propose une complétion de Knuth-Bendix qui permet donc d'obtenir un système fini et confluent, si la complétion de Knuth-Bendix termine avec succès. À partir d'un tel système, Maude permet, pour chaque terme, d'obtenir une forme normale unique. La comparaison de deux termes se ramène donc à la comparaison de leur forme normale.

Maude possède également une commande *xmatch* qui permet de faire du filtrage modulo les propriétés d'associativité, de commutativité et d'éléments neutres. Malheureusement, elle ne permet pas de le faire pour des propriétés définies par des équations quelconques.

Exemple 2.1.4
Voici un exemple très simple de spécification en Maude :

```
fmod EXEMPLE is
sort Matrix .
op 0 : -> Matrix [ctor] .
op I : -> Matrix [ctor] .
op _+_ : Matrix Matrix -> Matrix
                [prec 35 ctor assoc comm id: 0] .
op _*_ : Matrix Matrix -> Matrix
                [prec 25 ctor assoc id: I] .
vars N M P : Matrix .
eq 0 * N = 0 .
eq (N + P) * M = ( N * M ) + ( P * M ) .
eq M * (N + P) = ( M * N ) + ( M * P ) .
endfm
```

Dans cet exemple, sont définis :

- *une sorte : Matrix*
- *deux constantes : 0 et I*
- *deux opérateurs :*
  - *+ : associatif, commutatif et dont l'élément neutre est 0*
  - ** : associatif et dont l'élément neutre est I*

[1] http ://cime.lri.fr

- *des équations :*
  - *0 élément absorbant de la multiplication*
  - *distributivité de * par rapport à +.*

La commande

```
xmatch ( X:Matrix + Y:Matrix ) + Z:Matrix
       <=? a:Matrix + ( b:Matrix + c:Matrix ) .
```

qui cherche les valeurs de X, Y et Z, telles que $(X : Matrix + Y : Matrix) + Z : Matrix$ *soit égal à* $a : Matrix + (b : Matrix + c : Matrix)$ *renvoie 54 solutions parmi lesquelles :*

```
Solution 1
Matched portion = (whole)
Z:Matrix --> 0
X:Matrix --> 0
Y:Matrix --> a:Matrix + b:Matrix + c:Matrix

Solution 2
Matched portion = (whole)
Z:Matrix --> 0
X:Matrix --> a:Matrix
Y:Matrix --> b:Matrix + c:Matrix

Solution 10
Matched portion = (whole)
Z:Matrix --> a:Matrix
X:Matrix --> b:Matrix
Y:Matrix --> c:Matrix

Solution 11
Matched portion = (whole)
Z:Matrix --> a:Matrix
X:Matrix --> c:Matrix
Y:Matrix --> b:Matrix
```

Par contre,

```
xmatch ( X:Matrix * Y:Matrix) + ( Z:Matrix * U:Matrix)
       <=? a:Matrix * ( b:Matrix + c:Matrix ) .
```

renvoie 12 solutions faisant intervenir les éléments neutres, mais ne trouve pas :

```
X:Matrix --> a:Matrix
Y:Matrix --> b:Matrix
Z:Matrix --> a:Matrix
U:Matrix --> c:Matrix
```

2.1.3.3 Les limites de cette approche

Nous avons réalisé un premier prototype en utilisant Maude. Malheureusement, nous nous sommes vite heurtés à des problèmes liés à deux de nos hypothèses initiales : aucune

connaissance des technologies sous-jacentes (**R2**) et aucune connaissance sur le système (**R3**).

En effet la nécessité de fournir un ordre bien fondé pour utiliser la complétion de Knuth-Bendix impose à l'utilisateur un minimum de connaissances en théorie de la réécriture. Cet ordre n'est pas facile à trouver même pour un spécialiste du domaine. De nombreuses approches ont été étudiées pour engendrer automatiquement ou semi-automatiquement cet ordre mais aucune ne permet de couvrir tous les cas. Citons par exemple le *Recursive path ordering* introduit par Dershowitz dans [Der82]. De plus, la possibilité de composer des services complique notre problème, car une forme normale peut ne pas être suffisante pour résoudre notre problème.

En effet, étudions le cas où les services disponibles sont $serv1(A, B) = A + B$, $serv2(A, B) = A * B$ et $serv3(A, B, C, D) = A * B + C * D$. Supposons également que $*$ soit distributif par rapport à $+$. Si l'utilisateur souhaite réaliser $X * (Y + Z)$, plusieurs solutions sont envisageables, dont :

1. $serv3(X, Y, X, Z)$
2. $serv2(X, serv1(Y, Z))$
3. $serv1(serv2(X, Y), serv2(X, Z))$

Si la forme normale choisie est la forme factorisée, en comparant $X * (Y + Z)$ à $A * B + C * D$, la solution 1 ne sera pas trouvée. Si la forme normale choisie est la forme distribuée, en comparant $X * Y + X * Z$ à $A * B$, la solution 2 ne sera pas trouvée.

L'autre limite de cette approche est liée à l'algorithme de Knuth-Bendix qui :
– termine avec succès,
– termine avec échec (complétion impossible),
– ou ne termine pas car il engendre un ensemble infini de règles de réécriture
en fonction de la théorie de départ.

Notamment dès qu'une règle qui exprime la commutativité ($x + y \rightarrow y + x$) est introduite, le système ne peut pas être noéthérien.

Il était donc délicat, et en contradiction avec nos hypothèses de départ, de baser notre travail sur les techniques de réécriture, sachant que cela impose une connaissance de celles-ci pour le choix de l'ordre pour la complétion de Knuth-Bendix et impose des restrictions sur la théorie équationelle pour que la complétion de Knuth-Bendix termine avec succès.

Par contre, si nous nous plaçons dans le cadre d'une théorie figée pour un domaine particulier, un spécialiste peut construire l'ordre et générer un outil ad hoc qui sera modifié à chaque évolution de la théorie.

Notons également qu'il faut, pour engendrer tous les services, être capable d'obtenir tous les chemins de réécriture qui conduisent à la forme normale. En effet, ces différents chemins mèneront à une solution différente.

2.1.4 Unification, filtrage, unification équationnelle et filtrage équationnel

Une autre approche pour rechercher des correspondances entre les services fournis et la requête (comparaison de deux termes), en tenant compte d'un certain nombre d'égalités, est l'unification équationnelle et le filtrage équationnel.

Les problèmes de filtrage et d'unification, simple ou modulo une théorie équationnelle, avec ou sans type, appartiennent à une catégorie de problèmes qui ont été beaucoup

étudiés, car ils ont une place importante en informatique et notamment dans les assistants de preuve. Une bonne introduction à ces problèmes peut être trouvée dans [BS01].

2.1.4.1 L'unification

Le but de l'unification est de trouver une façon de rendre deux termes égaux en remplaçant certaines variables de ces deux termes.

Définition 2.1.23 (Unification)
Soient u et v, deux termes. Le problème d'*unification* est de savoir s'il existe une substitution σ, telle que $\sigma(u) = \sigma(v)$.
Un tel σ est appelé *unificateur* de u et v.

Exemple 2.1.5
Par exemple, si f est un symbole d'arité 2, a et b des symboles d'arité 0 et x et y des variables. Soient, $u = f(a, x)$ et $v = f(y, b)$, une solution au problème d'unification de u et v est $\sigma = \{x \to b, y \to a\}$: $\sigma(u) = f(a, b) = \sigma(v)$.

Il s'agit ici d'unification syntaxique du premier ordre, car les deux termes doivent être égaux et seules les variables (et pas les fonctions) peuvent être substituées.

Théorème 2.1.2
Le problème d'unification, du premier ordre, de deux termes est décidable.

Définition 2.1.24
Une substitution σ est un *mgu* (most general unifier) de u et v, s'il est un unificateur de u et v et si pour tout θ, unificateur de u et v, $\sigma \leq \theta$.

Théorème 2.1.3 ($mgus$)
L'ensemble des mgus *de deux termes peut être engendré à partir d'un unique* mgu. *Cela signifie que tous les* mgus *sont égaux modulo le renommage des variables.*

De nombreux algorithmes ont été proposés pour résoudre le problème d'unification. Il en existe en temps linéaire [MM82, PW76]. L'efficacité de l'algorithme proposé dans le second article est notamment lié à la représentation des termes sous forme de graphes acycliques orientés plutôt que sous forme d'arbre, ce qui permet de mutualiser le travail effectué sur les différents sous-termes identiques d'un même terme.

2.1.4.2 Le filtrage

Définition 2.1.25 (Filtrage)
Soient u et v deux termes, le problème de filtrage est de trouver σ tel que $\sigma(u) = v$. σ est appelé un *filtre* de u et v.

Exemple 2.1.6
Par exemple, si f est un symbole d'arité 2, a et b des symboles d'arité 0 et x une variable. Si $u = f(a, x)$ et $v = f(a, b)$, une solution au problème de filtrage de u et v est $\sigma = \{x \to b\}$: $\sigma(u) = f(a, b) = v$.

Si v ne contient pas de variable, les problèmes d'unification et de filtrage sont strictement identiques.

2.1.4.3 L'unification équationnelle

L'unification équationnelle se rapproche du problème d'unification mais au lieu de chercher à rendre les termes u et v égaux, nous cherchons à les rendre égaux modulo une congruence induite par une théorie équationnelle.

Définition 2.1.26 (E-unificateur)
Soient $\mathcal{E}$ une théorie équationnelle, u et v deux termes, σ est un *E-unificateur* de u et v si et seulement si $\sigma(u) =_{\mathcal{E}} \sigma(v)$.

Exemple 2.1.7
Par exemple, si f et g sont deux fonctions d'arité 2, a et b deux constantes, x et y deux variables et $\mathcal{E} = \{f(a,b) = g(a,b)\}$. Un E-unificateur de $u = f(a,x)$ et $v = g(y,b)$ est $\sigma = \{x \to a, y \to b\}$: $\sigma(u) = f(a,b) =_{\mathcal{E}} g(a,b) = \sigma(v)$.

Théorème 2.1.4
La propriété de décidabilité n'est pas conservée dans le cas d'une théorie équationnelle quelconque.

Définition 2.1.27 (Substitution plus générale modulo $\mathcal{E}$ et sur V)
Soient $\mathcal{E}$ une théorie équationnelle, V un ensemble de variables, une substitution σ est *plus générale modulo $\mathcal{E}$ et sur V* que θ ssi il existe une substitution δ telle que $\forall x \in V, \theta(x) =_{\mathcal{E}} \delta(\sigma(x))$.

Théorème 2.1.5
La propriété de l'unicité du mgu *n'est plus valide. Mais lorsqu'il sera fait référence à un ensemble complet d'E-unificateurs, c'est l'ensemble des* mgus *qui sera considéré.*

Théorème 2.1.6
L'ensemble minimal complet d'E-unificateurs peut ne pas exister et s'il existe, il peut être infini.

Du fait de la perte de la décidabilité et de l'unicité du *mgu* dans le cas général, de nombreux travaux ont été réalisés dans des théories particulières : associative [Mak77, Jaf90, Sch92], commutative [Sie79], distributive [AT85, SS96], associative et commutative [Dom91], ...

Des travaux ont également été réalisés pour pouvoir combiner les algorithmes de différentes théories [BS96]. La principale restriction de ces algorithmes est qu'ils nécessitent des théories sur des symboles disjoints. Cela limite donc les cas où ces techniques peuvent être utilisées.

2.1.4.4 Le filtrage équationnel

De la même façon que pour l'unification, où il existe l'unification équationnelle, il existe le problème de filtrage modulo une théorie équationnelle ou filtrage équationnel, aussi appelé E-matching.

Définition 2.1.28 (filtrage équationnel)
Le problème de *filtrage équationnel* de deux termes u et v a une solution si et seulement si il existe σ tel que $\sigma(u) =_{\mathcal{E}} v$.

Exemple 2.1.8
Par exemple, si f et g sont deux fonctions d'arité 2, a et b deux constantes, x une variable et $\mathcal{E} = \{f(a,b) = g(a,b)\}$. Une solution au problème de filtrage équationnel de $u = f(a,x)$ et $v = g(a,b)$ est $\sigma = \{x \to b\}$: $\sigma(u) = f(a,b) =_{\mathcal{E}} g(a,b) = v$.

Comme pour l'unification équationnelle, dans le cadre d'une théorie équationnelle quelconque, il n'y a pas forcément d'existence d'un ensemble minimal de filtres [FH86].

Là encore des travaux ont été réalisés dans le cas de théories particulières. Par exemple [Con04] pour une théorie associative et commutative, [MD96] pour des systèmes de réécriture ou [DMS92] pour des systèmes convergents.

2.1.4.5 Et les types ?

Tous ces problèmes existent également dans une version typée. L'ajout de types implique des contraintes supplémentaires sur les substitutions calculées, car les variables doivent être remplacées par des termes avec des « types compatibles ».

Parmi les travaux réalisés, citons [SNGM89, MGS89]. Ces travaux se placent dans le cadre de types ayant des liens d'héritage. Pour intégrer les types à l'unification, il y a deux solutions : soit réaliser une phase d'unification simple sans tenir compte des types, suivie par une phase de typage, soit intégrer le typage à chaque étape de l'unification. Cette seconde solution est, en général, plus efficace car elle permet de détecter les erreurs de types plus tôt. Cependant, elle peut entraîner la duplication du travail pour tous les types possibles.

Dans [Kir88], Claude Kirchner étudie l'unification équationnelle dans le cas d'algèbres avec types qui ont des liens d'héritage. Dans le cadre de théories régulières (chaque terme à un unique plus petit type) et qui conservent les types (tous les termes égaux ont le même plus petit type), les règles de l'unification standard sont étendues dans le cas typé.

Nous avons justifié dans la section 1.1 que nous devions pouvoir rechercher des services quelle que soit la théorie équationnelle définie. Il n'est donc pas possible de s'appuyer sur les résultats décidables dédiés à des théories particulières. Nous avons donc choisi de nous appuyer sur l'algorithme général proposé par Gallier et Snyder dans [GS89]. En effet, nous allons voir dans la section suivante que la définition de notre ensemble de solutions nous ramène à un problème de filtrage équationnel entre les services et la requête.

2.2 Représentation du domaine, des services et de la requête

Nous venons de détailler plus précisément ce qu'est une spécification algébrique. Nous avons choisi de représenter le domaine grâce à une telle spécification. Les services et la requête sont alors exprimés grâce à des termes de l'algèbre associée à la signature.

2.2.1 Le domaine

Comme nous venons de le voir, nous avons choisi de définir un domaine par une signature hétérogène avec sous-typage (S,$\leq$,Σ) et un ensemble d'équations $\mathcal{E}$.

Dans la suite, tous les exemples seront développés sur un domaine simple défini comme suit :

Définition 2.2.1 (Définition du domaine des exemples)
$S = \{Matrix\}$

$$
\begin{aligned}
&\Sigma = \{ \\
&\qquad 0 : \rightarrow Matrix \\
&\qquad I : \rightarrow Matrix \\
&\qquad + : Matrix \times Matrix \rightarrow Matrix \\
&\qquad * : Matrix \times Matrix \rightarrow Matrix \\
&\} \\
&\mathcal{E} = \{ \\
&\qquad x : Matrix \qquad x * I = x \\
&\qquad x : Matrix \qquad I * x = x \\
&\qquad x : Matrix \qquad x * 0 = 0 \\
&\qquad x : Matrix \qquad 0 * x = 0 \\
&\qquad x, y : Matrix \qquad x + y = y + x \\
&\qquad x : Matrix \qquad x + 0 = x \\
&\qquad x, y, z : Matrix \qquad x + (y + z) = (x + y) + z \\
&\qquad x, y, z : Matrix \qquad x * (y * z) = (x * y) * z \\
&\qquad x, y, z : Matrix \qquad x * (y + z) = (x * y) + (x * z) \\
&\qquad x, y, z : Matrix \qquad (x + y) * z = (x * z) + (y * z) \\
&\}
\end{aligned}
$$

2.2.2 Les services

Le problème à résoudre est de trouver l'ensemble des services et des compositions de services répondant à une requête. Pour cela, nous disposons d'un ensemble de bibliothèques proposant un ensemble de services. Nous noterons S l'ensemble de ces services, appelés *services initiaux*.

Pour décrire les services, nous nous appuyons sur le λ-calcul, qui est une abstraction du mécanisme de déclaration et d'appel de fonctions. La syntaxe du λ-calcul est :

$$M ::= x \mid MM \mid \lambda x.M$$

où :

- x est une variable
- $\lambda x.M$ est une abstraction (ou fonction de paramètre formel x et de corps M)
- MN est l'application de M à N, en particulier $(\lambda x.M)N$ est un appel de fonction, où le paramètre formel x est remplacé par le paramètre réel N.

Cet appel de fonction est représenté par la règle de réduction :

$$(\lambda x.M)N \rightarrow \sigma(M), \text{ où } \sigma = \{x \mapsto N\}$$

Cette règle de réduction est appelée β-réduction.

Un service initial s possède un ensemble de paramètres noté P_s. La fonctionnalité rendue est exprimée par un terme de l'algèbre.

Soit $\{p_1, ..., p_n\}$ l'ensemble des variables de P_s qui interviennent dans la description de la fonctionnalité. Un service s est alors de la forme $\lambda p_1 ... \lambda p_n.t_s$, avec t_s un terme appartenant à $T_\Sigma[\{p_1, ..., p_n\}]$. Dans la suite, $\{p_1, ..., p_n\}$ sera noté $Var(t_s)$ et $\lambda p_1 ... \lambda p_n$ sera noté $\lambda Var(t_s)$.

Exemple 2.2.1
Exemples de services :
$s_1 = \lambda x \lambda y \lambda z.x + (y + z)$
$s_2 = \lambda x \lambda y.x * y$
$s_3 = \lambda x \lambda y \lambda z.x * (y + z)$
$s_4 = \lambda x \lambda y \lambda z \lambda u \lambda w.x * (y * w) + z * u$

Les éléments de P_s qui ne sont pas dans $\{p_1, ..., p_n\}$ sont des paramètres dont nous ne saurons pas déterminer les valeurs. Ce sont ceux qui n'interviennent pas directement dans la fonctionnalité du service. Dans les solutions retournées, leur valeur sera affectée à « ? ». Ils devront être traités « à la main » par l'utilisateur. Il est aussi envisageable de coupler nos algorithmes de courtage avec un autre outil permettant de déterminer ces paramètres.

Exemple 2.2.2
Soit le service s de signature :
$s(A : Matrix,\ B : Matrix,\ m : int,\ n : int,\ p : int) : Matrix$ *qui réalise le produit de la matrice A de taille $m \times n$ avec la matrice B de taille $n \times p$.*

Dans notre représentation, s pourra par exemple être représenté par :
$\lambda A \lambda B.A * B.$

Dans ces conditions, les paramètres m, n et p ne pourront pas être déterminés par notre analyse.

2.2.3 La requête

L'utilisateur a un problème particulier qu'il veut résoudre à partir de ses propres données. Pour cela, il peut appeler un ou des services avec ses données en paramètres. C_r est l'ensemble des identificateurs qui permettent de désigner ses données. Il va alors exprimer son problème sous la forme d'une requête r qui est un terme appartenant à $T_\Sigma[C_r]$. Dans la suite, les éléments de C_r seront appelés *constantes de la requête*.

Exemple 2.2.3
Exemples de requêtes :

$C_{r_1} = \{a, b : Matrix \quad\}, r_1 = a + b$
$C_{r_2} = \{a, b : Matrix \quad\}, r_2 = a * b$
$C_{r_3} = \{a, b : Matrix \quad\}, r_3 = a * (b + I)$
$C_{r_4} = \{a, b, c, d : Matrix\}, r_4 = a * b + c * d$

2.3 Description de l'ensemble des services réalisables

Maintenant que nous avons décrit la requête et les services, nous souhaitons formaliser l'ensemble des services ou des compositions de services qui permettent de répondre à la requête de l'utilisateur. Pour cela, nous devons décrire l'ensemble des services et des compositions de services qui peuvent être rendus à partir des services initiaux.

À partir de l'ensemble des services initiaux S, nous pouvons engendrer d'autres services par affectations de valeurs aux paramètres ou par transformations en appliquant des équations. Dans le dernier cas, le service rendu est le même mais la forme sous laquelle il est exprimé est différente.

Plus formellement, définissons l'ensemble des transformations sur les services. Dans la suite, t_s et t sont des termes sur Σ, les x_i et y_i sont des variables, c est une constante, α une position dans un terme et σ une substitution.

- $cste(x, c) : \lambda x \lambda x_1 ... \lambda x_n.t_s \xrightarrow{cste(x,c)} \lambda x_1 ... \lambda x_n.\sigma(t_s)$, où :
  - $\sigma = \{x \mapsto c\}$;
  - c est une constante du domaine ou de la requête.

  La transformation $cste$ fixe la valeur d'un paramètre à une constante du domaine ou de la requête. Le service obtenu est moins général que celui dont il est issu. Le paramètre affecté est supprimé de la liste des paramètres du nouveau service et il est remplacé par la constante dans la fonctionnalité rendue.

- $comp(x, t) : \lambda x \lambda x_1 ... \lambda x_n.t_s \xrightarrow{comp(x,t)} \lambda x_1 ... \lambda x_n \lambda y_1 ... \lambda y_m.\sigma(t_s)$, où :
  - $\sigma = \{x \mapsto t\}$;
  - t est un terme ;
  - $\{y_1, ..., y_m\} = Var(t) \backslash \{x_1, ..., x_n\}$.

  La transformation $comp$ fixe la valeur d'un paramètre à un terme. Cette règle représente la composition de services. Dans la suite, nous ferons l'hypothèse que le terme affecté est un service réalisable à partir des services initiaux et que les variables de ce terme sont des nouvelles variables (au besoin les variables seront renommées). Le paramètre affecté est supprimé de la liste des paramètres, les variables du terme sont ajoutées comme nouveaux paramètres et le paramètre est remplacé par le terme dans la fonctionnalité.

- $lie(x, y)$:

  Si $\exists k$ tel que $y = x_k$, alors $\lambda x \lambda x_1 ... \lambda x_n.t_s \xrightarrow{lie(x,y)} \lambda x_1 ... \lambda x_n.\sigma(t_s)$

  Sinon y doit être une nouvelle variable et $\lambda x \lambda x_1 ... \lambda x_n.t_s \xrightarrow{lie(x,y)} \lambda y \lambda x_1 ... \lambda x_n.\sigma(t_s)$, où $\sigma = \{x \mapsto y\}$.

  La transformation lie a deux significations. Soit elle renomme un paramètre avec un nouveau nom, soit elle lie deux paramètres. Dans le cas où elle lie deux paramètres, le nouveau service est formé en spécifiant que les deux paramètres doivent avoir la même valeur. Un des paramètres est supprimé et il est remplacé par l'autre paramètre dans la fonctionnalité du service.

- $eq(\alpha, \sigma, e_1, e_2)$:

  si $t_{s|_\alpha} = \sigma(e_1)$ alors $\lambda x_1 ... \lambda x_n.t_s \xrightarrow{eq(\alpha,\sigma,e_1,e_2)} \lambda x_{i_1} ... \lambda x_{i_l} \lambda y_1 ... \lambda y_m.t_s[\alpha \leftarrow \sigma(e_2)]$, où :
  - $\alpha \in \mathcal{O}(t)$;
  - $t_{|_\alpha}$ désigne le sous-terme de t à la position α .
  - $t[\alpha \leftarrow s]$ représente le terme t dans lequel le sous-terme à la position α a été remplacé par s.
  - (e_1, e_2) est une variante d'une équation de $\mathcal{E}$, avec des nouvelles variables, c'est-à-dire qu'il existe une substitution δ inversible, telle que $(\delta^-1(e_1), \delta^-1(e_2)) \in \mathcal{E}$.
  - $\forall v \in Im(\sigma|_{Var(e_2) \backslash Var(e_1)}),\ v \in Var(t_s)$ ou v est une nouvelle variable ;

– $\{y_1, ..., y_m\} = Var(\sigma(e_2))\backslash Var(\sigma(e_1))$;
– $\{x_{i_1}, ..., x_{i_l}\} = \{x_1, ...x_n\}\backslash(Var(\sigma(e_1))\backslash Var(\sigma(e_2)))$.

La transformation eq n'a aucun impact sur le service rendu, seule la forme sous laquelle il est exprimé diffère. La fonctionnalité est remplacée par une fonctionnalité équivalente en appliquant une équation. Nous faisons disparaître les paramètres qui sont supprimés par l'application de l'équation et apparaître ceux introduits par l'équation. La contrainte sur l'image de σ est présente pour s'assurer que les seules variables qui peuvent être introduites dans le problème sont des nouvelles variables et qu'il est impossible de faire réapparaître une variable qui a déjà été utilisée.

Bien entendu, nous faisons l'hypothèse forte que les équations définies par les spécialistes d'un domaine sont bien des équivalences qui préservent la sémantique.

Propriété 2.3.1
Soient t_1, t_2 des termes sur Σ et A une suite de transformations telle que :

$$\lambda Var(t_1).t_1 \xrightarrow{A} \lambda X.t_2$$

Alors $X = Var(t_2)$.

Cette propriété nous permettra par la suite de ne pas développer l'ensemble de variables sous le lambda quand il sera trop compliqué. Nous noterons alors $\lambda.t$ au lieu de $\lambda Var(t).t$.

Preuve Par induction sur la taille de A. Triviale, par définition, pour chaque transformation. □

Corollaire 2.3.1
Les services initiaux étant de la forme $\lambda Var(t_s).t_s$, avec t_s un terme sur Σ, tous les services engendrés seront de la forme $\lambda Var(t).t$, avec t un terme sur Σ.

Remarque : Ce corollaire montre également qu'il n'était pas nécessaire d'utiliser des λ-termes pour représenter les services, des termes auraient été suffisants. Néanmoins, ce choix à été réalisé pour faire apparaître explicitement les paramètres de ces services.

Exemple 2.3.1
Exemple de production de services :

$$\begin{array}{ll} s = & \lambda x\lambda y\lambda z\lambda u.x * y + z * u \\ \xrightarrow{cste(u,0)} & \lambda x\lambda y\lambda z.x * y + z * 0 \\ \xrightarrow{eq(2,\{x_1 \mapsto z\},x_1 * 0,0)} & \lambda x\lambda y.x * y + 0 \\ \xrightarrow{eq(\epsilon,\{x_2 \mapsto x * y\},x_2 + 0,x_2)} & \lambda x\lambda y.x * y \end{array}$$

Exemple 2.3.2
Exemple de production de services :

$$\begin{array}{ll} s = & \lambda x\lambda y.x * y \\ \xrightarrow{comp(y,a+b)} & \lambda x\lambda a\lambda b.x * (a + b) \\ \xrightarrow{eq(\epsilon,\{x_1 \mapsto x;x_2 \mapsto a;x_3 \mapsto a\},x_1 * (x_2 + x_3),x_1 * x_2 + x_1 * x_3)} & \lambda x\lambda a\lambda b.x * a + x * b \end{array}$$

Exemple 2.3.3
Exemple de production de services :

$$
\begin{array}{ll}
s = & \lambda x\lambda y\lambda z.x*y+z \\
\xrightarrow{cste(z,0)} & \lambda x\lambda y.x*y+0 \\
\xrightarrow{eq(2,\emptyset,0,x_1*0)} & \lambda x\lambda y\lambda x_1.x*y+x_1*0 \\
\xrightarrow{lie(x_1,x)} & \lambda x\lambda y.x*y+x*0 \\
\xrightarrow{eq(\epsilon,\{x_2\mapsto x;x_3\mapsto y;x_4\mapsto 0\},x_2*x_3+x_2*x_4,x_2*(x_3+x_4))} & \lambda x\lambda y.x*(y+0)
\end{array}
$$

Définition 2.3.1 (Ensemble des services pouvant être rendus)
Soit S l'ensemble des services initiaux.
Notons $S_i = \{\lambda Var(t).t|\ \exists s \in S,\ t_1;...;t_n$ une suite de transformations de taille n telle que : $\lambda Var(s).s \xrightarrow{t_1;...;t_n} \lambda Var(t).t\}$.
L'*ensemble des services pouvant être rendus* est $\bigcup_{i=0}^{\infty} S_i$.

2.4 Description de l'ensemble des solutions

Nous venons de construire l'ensemble des services et des compositions de services réalisables à partir des services initiaux. Une requête r sera réalisable si elle appartient à cet ensemble.

Définition 2.4.1 (Répondre à la requête)
Dans le cas où les compositions de services sont interdites, un service initial $s = \lambda Var(t_s).t_s$ permet de *répondre à la requête* r, s'il existe une suite A de transformations telle que :

- il n'y a pas de transformation $comp$ dans A ;
- $\lambda Var(t_s).t_s \xrightarrow{A} r$.

Exemple 2.4.1
$\lambda x\lambda y\lambda z\lambda u.x*y+z*u$ *permet de répondre à* $a+b$

$$
\begin{array}{ll}
 & \lambda x\lambda y\lambda z\lambda u.x*y+z*u \\
\xrightarrow{cste(x,I)} & \lambda y\lambda z\lambda u.I*y+z*u \\
\xrightarrow{cste(z,I)} & \lambda y\lambda u.I*y+I*u \\
\xrightarrow{eq(1,\{x_1\mapsto y\},I*x_1,x_1)} & \lambda y\lambda u.y+I*u \\
\xrightarrow{eq(2,\{x_2\mapsto u\},I*x_2,x_2)} & \lambda y\lambda u.y+u \\
\xrightarrow{cste(y,a)} & \lambda u.a+u \\
\xrightarrow{cste(u,b)} & a+b
\end{array}
$$

Dans le cas où la composition de services est autorisée, un service initial $s = \lambda Var(t_s).t_s$ permet de répondre à la requête r, s'il existe une suite A de transformations telle que :

- pour toutes les transformations $comp(x_i, t_i)$ de A, le problème t_i peut être rendu à l'aide des services de S ;
- $\lambda Var(t_s).t_s \xrightarrow{A} r$.

Exemple 2.4.2
Soit $S = \{\lambda x\lambda y.x+y;\ \lambda x\lambda y.x*y\}$.

$\lambda x \lambda y.x + y$ *permet de répondre à* $a + (b * c)$ *car :*

$$\lambda x \lambda y.x + y$$
$$\xrightarrow{cste(x,a)} \quad \lambda y.a + y$$
$$\xrightarrow{comp(y,b*c)} \quad a + (b * c)$$

et

$$\lambda x \lambda y.x * y$$
$$\xrightarrow{cste(x,b)} \quad \lambda y.b * y$$
$$\xrightarrow{comp(y,c)} \quad b * c$$

Nous remarquons ici que la définition précédente conduit à des solutions potentiellement infinies. Ce problème sera pris en compte dans nos algorithmes en imposant une profondeur maximale de composition.

L'existence de la suite de transformations A implique qu'il existe une suite d'affectations des paramètres (à des constantes, à des variables ou à des termes dans le cadre de la composition) et de transformations de la représentation du service, qui permet de transformer t_s en r. Les liens entre les paramètres du service et les constantes du domaine et de la requête seront mis en évidence à partir de cette suite de transformations. Ceux-ci seront exprimés sous forme de substitutions. Ces substitutions, appliquées sur les paramètres du service, donneront les valeurs qu'il faut associer à ces paramètres lors de l'appel du service.

Définition 2.4.2 (σ_A)
À toute suite de transformations A est associée une substitution σ_A engendrée à partir des transformations suivantes :

- $\sigma_{cste(x,c)} = \{x \mapsto c\}$
- $\sigma_{comp(x,t)} = \{x \mapsto t\}$
- $\sigma_{lie(x,y)} = \{x \mapsto y\}$
- $\sigma_{eq(\alpha,\sigma,e_1,e_2)} = \emptyset$
- $\sigma_{A_1;A_2} = \sigma_{A_1} \circ \sigma_{A_2}$

Justifications 1
Explications informelles de ces substitutions :

- $\sigma_{cste(x,c)} = \{x \mapsto c\}$
  Appliquer $cste$ revient à transformer le service en remplaçant un de ses paramètres par c. Ce lien se retrouve dans la substitution.
- $\sigma_{comp(x,t)} = \{x \mapsto t\}$
  Appliquer $comp$ revient à transformer le service en remplaçant un de ses paramètres par t. Ce lien se retrouve dans la substitution. Pour que la solution soit valide, t doit être réalisable à l'aide des services de S. Dans la pratique, une variable intermédiaire sera associée à t. Elle recevra le résultat du traitement de t et c'est elle qui sera passée en paramètre du service. Cette réalisation de t sera exécutée avant le service qui prend t en paramètre.
- $\sigma_{lie(x,y)} = \{x \mapsto y\}$
  Si y n'est pas une nouvelle variable, appliquer lie signifie que deux paramètres sont liés. Le lien entre le nom du paramètre supprimé et le nom du paramètre conservé doit donc apparaître dans la substitution. Sinon nous sommes dans le cas du renommage d'un paramètre, le lien entre l'ancien nom et le nouveau nom doit apparaître dans la substitution.

- $\sigma_{eq(\alpha,\sigma,e_1,e_2)} = \emptyset$
  Il n'y a aucune affectation des paramètres. Seule la forme sous laquelle est exprimée la fonctionnalité du service change. Cela n'aura donc pas d'impact sur la substitution.
- $\sigma_{A_1;A_2} = \sigma_{A_1} \circ \sigma_{A_2}$
  Des paramètres introduits ou associés dans la première partie des transformations peuvent être affectés dans la seconde. σ_{A_2} *et* σ_{A_1} *sont donc composées.*

Exemple 2.4.3
La substitution associée à l'exemple 2.4.1 (page 47) est :
$\{x \mapsto I,\ z \mapsto I,\ y \mapsto a,\ u \mapsto b\}$.

Exemple 2.4.4
La substitution associée à l'exemple 2.3.3 (page 47) est :
$\{z \mapsto 0,\ x_1 \mapsto x\}$.

Propriété 2.4.1
Soient a *et* b *deux termes de* T_Σ *et* A *une suite de transformations :*
Si $\lambda Var(a).a \xrightarrow{A} \lambda Var(b).b$
Alors $\sigma_A(a) =_\mathcal{E} b$.

Preuve Par récurrence sur la taille de A.

- Cas d'arrêt :
  Si A est de taille 1, traitons les différentes transformations :
  - $\lambda Var(a).a \xrightarrow{cste(x,c)} \lambda Var(b).b$
    Par définition de $cste$: $b = \{x \mapsto c\}(a)$,
    or $\sigma_A = \{x \mapsto c\}$,
    donc $\sigma_A(a) = b$
    et donc $\sigma_A(a) =_\mathcal{E} b$.
  - $\lambda Var(a).a \xrightarrow{comp(x,t)} \lambda Var(b).b$
    Par définition de $comp$: $b = \{x \mapsto t\}(a)$,
    or $\sigma_A = \{x \mapsto t\}$,
    donc $\sigma_A(a) = b$
    et donc $\sigma_A(a) =_\mathcal{E} b$.
  - $\lambda Var(a).a \xrightarrow{lie(x,y)} \lambda Var(b).b$
    Par définition de lie : $b = \{x \mapsto y\}(a)$,
    or $\sigma_A = \{x \mapsto y\}$,
    donc $\sigma_A(a) = b$
    et donc $\sigma_A(a) =_\mathcal{E} b$.
  - $\lambda Var(a).a \xrightarrow{eq(\alpha,\theta,e_1,e_2)} \lambda Var(b).b$
    Par définition de eq : $a =_\mathcal{E} b$
    or $\sigma_A = \emptyset$,
    et donc $\sigma_A(a) =_\mathcal{E} b$.

  La propriété est donc vraie.

- Poursuite :
  Supposons que la propriété soit vraie pour une transformation de taille inférieure à n, montrons qu'elle est vraie pour une transformation de taille n.

Soient A_1 et A_2 deux suites de transformations non vide et c un terme sur Σ tels que A = A_1 ;A_2 et $\lambda Var(a).a \xrightarrow{A_1} \lambda Var(c).c \xrightarrow{A_2} \lambda Var(b).b$.
Par hypothèse de récurrence :
– $\sigma_{A_1}(a) =_{\mathcal{E}} c$
– $\sigma_{A_2}(c) =_{\mathcal{E}} b$
donc $\sigma_{A_2}(\sigma_{A_1}(a)) =_{\mathcal{E}} \sigma_{A_2}(c) =_{\mathcal{E}} b$.
Or, par définition, $\sigma_A = \sigma_{A_1;A_2} = \sigma_{A_1} \circ \sigma_{A_2}$, donc $\sigma_A(a) =_{\mathcal{E}} b$.
La propriété est donc vraie.

□

Nous reconnaissons ici la notion de filtrage équationnel.

Corollaire 2.4.1
Soient a et b deux termes sur Σ et A une suite de transformations sans transformation « eq » :
Si $\lambda Var(a).a \xrightarrow{A} \lambda Var(b).b$
Alors $b = \sigma_A(a)$.

Preuve Immédiate en reprenant la preuve précédente. □

Définition 2.4.3 (Ensemble des solutions)
Soit r une requête l'ensemble des solutions répondant à la requête r, est l'ensemble :

$$\{(s, \sigma_A) \mid s \in S, \lambda Var(s).s \xrightarrow{A} r\}$$

Justifications 2
D'après la propriété précédente, en appliquant σ_A sur la fonctionnalité du service, le service est égal, modulo les équations, à la requête. Donc cette substitution va indiquer les valeurs des paramètres avec lesquelles le service doit être appelé pour exécuter la requête. Dans le cas où une composition est nécessaire, le terme correspondant au sous-service sera construit avant l'appel de s.

2.5 Propriétés des suites de transformations

Ré-ordonnancement Nous allons maintenant montrer que ces suites de transformations peuvent être réordonnées pour, dans un premier temps, appliquer la substitution (grâce à une suite de transformations $cste$, lie et $comp$) puis dans un second temps appliquer les équations (grâce à une suite de transformations eq).

Définition 2.5.1 (A_δ)
À toute substitution δ, nous associons une suite de transformations A_δ engendrée comme suit :
– $A_{\{x \mapsto r\}}$ = $cste(x, r)$, si r est une constante du domaine ou de la requête ;
– $A_{\{x \mapsto r\}}$ = $lie(x, r)$, si r est une variable ;
– $A_{\{x \mapsto r\}}$ = $comp(x, r)$, si r est un terme ;
– $A_{\sigma_1 \cup \sigma_2}$ = A_{σ_1}; A_{σ_2}.

Nous remarquons que, pour toute substitution δ, aucune transformation eq n'apparaît dans la suite de transformations A_δ .

Propriété 2.5.1
Soient $\{x_1, ..., x_n\}$ un ensemble de variables et $\{r_1, ..., r_n\}$ un ensemble de termes. Soit $\delta = \{x_1 \mapsto r_1, ..., x_n \mapsto r_n\}$ alors $\sigma_{A_\delta} = \delta$ si $Dom(\delta) \cap Im(\delta) = \emptyset$.

Preuve Par récurrence sur la taille de la substitution.

- Cas d'arrêt :
  Si la substitution δ est de taille 1, notons $\delta = \{x \mapsto r\}$
  - si r est une constante du domaine ou de la requête, $A_\delta = cste(x, r) : \sigma_{A_\delta} = \sigma_{cste(x,r)} = \{x \mapsto r\} = \delta$
  - si r est une variable, $A_\delta = lie(x, r) : \sigma_{A_\delta} = \sigma_{lie(x,r)} = \{x \mapsto r\} = \delta$
  - si r est un terme, $A_\delta = comp(x, r) : \sigma_{A_\delta} = \sigma_{comp(x,r)} = \{x \mapsto r\} = \delta$
- Poursuite :
  Si la propriété est vraie pour une substitution de taille inférieure à n, montrons que la propriété est vraie pour une substitution de taille n.

  $\delta = \{x_1 \mapsto r_1\} \cup \delta_1$
  $\sigma_{A_\delta} = \sigma_{A_{\{x_1 \mapsto r_1\} \cup \delta_1}}$
  $= \sigma_{A_{\{x_1 \mapsto r_1\}; A_{\delta_1}}}$
  $= \sigma_{A_{\{x_1 \mapsto r_1\}}} \circ \sigma_{A_{\delta_1}}$
  $= \{x_1 \mapsto r_1\} \circ \delta_1$ par hypothèse de récurrence $\sigma_{A_{\delta_1}} = \delta_1$
  $= \delta_1 \cup \{x_1 \mapsto r_1\}$ car $x_1 \notin Im(\delta_1)$, $x_1 \notin Dom(\delta_1)$ et $r_1 \notin Dom(\delta_1)$ puisque δ est une substitution
  $= \delta$

  La propriété est donc vraie. □

Propriété 2.5.2
Une variable qui disparaît dans une suite de transformations ne peut pas réapparaître.

Pour toutes suites de transformations A_1 et A_2 et tous termes t_1, t_2 et t_3 tels que : $\lambda Var(t_1).t_1 \xrightarrow{A_1} \lambda Var(t_2).t_2 \xrightarrow{A_2} \lambda Var(t_3).t_3$.

Si $v \in Var(t_1)$ et $v \notin Var(t_2)$ alors $v \notin Var(t_3)$.

Preuve Les transformations qui permettent d'introduire des variables dans les termes sont $comp$, lie et eq. En ce qui concerne $comp$ et lie, ces variables doivent être des nouvelles variables, c'est-à-dire qu'elles ne doivent pas être apparues précédemment dans le problème.

En ce qui concerne eq, la contrainte :

$$\forall v \in Im(\sigma|_{Var(e_2) \setminus Var(e_1)}),\ v \in Var(t_s) \text{ ou } v \text{ est une nouvelle variable}$$

impose que les variables potentiellement introduites par l'équation sont soit déjà présentes dans le terme (elles n'ont donc pas disparu), soit des nouvelles variables.

Donc les seules variables introduites sont des nouvelles variables. □

Propriété 2.5.3
Toutes les substitutions engendrées à partir d'une suite de transformations sont idempotentes.

Preuve D'après la propriété 2.1.1 (page 34), les substitutions sont idempotentes si et seulement si les domaines et les images des substitutions ont une intersection nulle.

Soient A une suite de transformations et σ_A la substitution engendrée à partir de A. Faisons une preuve par récurrence sur la taille de A.

Cas d'arrêt :

Pour une transformation vide, la substitution associée est l'identité, donc la propriété est vraie.

Poursuite :

Supposons que la propriété est vraie pour une suite de transformations de taille $n-1$, montrons qu'elle est vraie pour une suite de transformations de taille n.

Nous allons montrer que si $A = A_1; t$, où A_1 est une suite transformations de taille $n-1$ et t une transformation, alors :

$$Dom(\sigma_{A_1}) \cap Im(\sigma_{A_1}) = \emptyset \Rightarrow Dom(\sigma_A) \cap Im(\sigma_A) = \emptyset$$

1. $A = A_1; eq(\alpha, \sigma, e_1, e_2)$

   $\sigma_A = \sigma_{A_1}$ donc la propriété est vraie par hypothèse de récurrence.

2. $A = A_1; cste(s, c)$ ou $A = A_1; lie(s, r)$ ou $A = A_1; comp(s, t)$

   Notons X pour c ou r ou t. Par définition, $\sigma_A = \sigma_{A_1} \circ \{s \mapsto X\}$. Par hypothèse de récurrence : $Dom(\sigma_{A_1}) \cap Im(\sigma_{A_1}) = \emptyset$.

   - Si $s \in Dom(\sigma_{A_1})$, comme $Dom(\sigma_{A_1}) \cap Im(\sigma_{A_1}) = \emptyset$, alors $s \notin Im(\sigma_{A_1})$. Donc $\sigma_A = \sigma_{A_1}$, la propriété est vérifiée.
   - Si $s \in Im(\sigma_{A_1})$, comme $Dom(\sigma_{A_1}) \cap Im(\sigma_{A_1}) = \emptyset$, alors $s \notin Dom(\sigma_{A_1})$.
     Nous avons :
     - $Dom(\sigma_A) = Dom(\sigma_{A_1}) \cup \{s\}$
     - $Im(\sigma_A) = (Im(\sigma_{A_1}) \backslash \{s\}) \cup Var(X)$

     Traitons les trois cas pour X :
     - Si $X = c$, alors $Var(X) = \emptyset$, la propriété est vérifiée.
     - Si $X = r$.
       - Si r est une nouvelle variable, alors $r \notin Dom(\sigma_{A_1})$, la propriété est donc vraie.
       - Si r apparaît dans le terme sur lequel la transformation est appliquée.
         Preuve par l'absurde.
         Supposons que $r \in Dom(\sigma_{A_1})$ et montrons que nous arrivons à une contradiction.
         Il existe alors une transformation de la forme $cste(r, c')$ ou $lie(r, r')$ ou $comp(r, t')$ dans A_1. Après application d'une telle transformation r n'apparaît plus dans le terme. r ne pouvant pas être réintroduit d'après la propriété 2.5.2 (page 51), elle n'appartient donc pas au terme sur lequel est appliquée la transformation.
         Il y a donc contradiction, donc $r \notin Dom(\sigma_{A_1})$ et la propriété est vérifiée.
       - Si $X = t$, par définition de la règle $comp$, $Var(t)$ n'est composé que de nouvelles variables, elles ne sont donc pas dans $Dom(\sigma_{A_1})$, la propriété est donc vraie.
   - Sinon : $s \notin Dom(\sigma_{A_1})$ et $s \notin Im(\sigma_{A_1})$.
     Nous avons :
     - $Dom(\sigma_A) = Dom(\sigma_{A_1}) \cup \{s\}$
     - $Im(\sigma_A) = Im(\sigma_{A_1}) \cup Var(X)$

     Par hypothèse, $s \notin Im(\sigma_{A_1})$, par définition de $cste$, lie et $comp$ $s \notin Var(X)$ et pour les mêmes raisons que ci-dessus $Var(X) \cap Dom(\sigma_{A_1}) = \emptyset$. La propriété est donc vérifiée.

La propriété est donc vraie et par récurrence, elle est également vraie pour toute suite de transformations.

□

Corollaire 2.5.1
Pour toute suite de transformations A, σ_A (la substitution engendrée à partir de A) et $\sigma_{A_{\sigma_A}}$ (la substitution engendrée à partir de la transformation générée à partir de σ_A) sont identiques : $\sigma_A = \sigma_{A_{\sigma_A}}$.

Preuve D'après la propriété 2.5.1 (page 51), si $Dom(\delta) \cap Im(\delta) = \emptyset$, alors $\sigma_{A_\delta} = \delta$. Nous venons de montrer que, pour toute suite de transformations A, $Dom(\sigma_A) \cap Im(\sigma_A) = \emptyset$ donc $\sigma_{A_{\sigma_A}} = \sigma_A$. □

Lemme 2.5.1
Soient s et r deux termes sur Σ.
Si $\lambda Var(s).s \xrightarrow{A} \lambda Var(r).r$,
alors $\lambda Var(\sigma_A(s)).\sigma_A(s) \xrightarrow{A_E} \lambda Var(r).r$,
où A_E est une variante de la suite des transformations eq de A.

Preuve La preuve se décompose en 4 étapes :
- décomposition de la transformation initiale ;
- vérification de la condition d'égalité des sous-termes pour appliquer la transformation eq ;
- vérification de la condition sur l'image de θ pour appliquer la transformation eq ;
- vérification que le résultat de la transformation est bien r.

1. A est une suite de transformations $cste$, $comp$, lie et eq. Nous la découpons de façon à isoler les transformations eq. Nous obtenons alors :
   $A = A_1; eq(\alpha_1, \theta_1, e_{11}, e_{21}); A_2; ...; eq(\alpha_{n-1}, \theta_{n-1}, e_{1n-1}, e_{2n-1}); A_n$, les A_i ne contiennent aucune transformation eq et peuvent être vides.

   Nous avons alors :

$$\begin{array}{ll} & \lambda.s \\ \xrightarrow{A_1} & \lambda.\sigma_{A_1}(s) \\ \xrightarrow{eq(\alpha_1,\theta_1,e_{11},e_{21})} & \lambda.s_1 \\ \xrightarrow{A_2} & \lambda.\sigma_{A_2}(s_1) \\ & \dots \\ \xrightarrow{eq(\alpha_{n-1},\theta_{n-1},e_{1n-1},e_{2n-1})} & \lambda.s_{n-1} \\ \xrightarrow{A_n} & \lambda.\sigma_{A_n}(s_{n-1}) = r \end{array}$$

   Avec, par définition, $\sigma_A = \sigma_{A_1} \circ ... \circ \sigma_{A_n}$.

2. Les substitutions s'appliquant sur les variables qui sont les feuilles des termes, elles n'agissent pas sur les positions des sous-termes : $\forall t,\ \sigma(t_{|_\alpha}) = \sigma(t)_{|_\alpha}$.

   (a) Traitement de la première équation

   Comme la transformation $eq(\alpha_1, \theta_1, e_{11}, e_{21})$ est applicable : $\sigma_{A_1}(s)_{|_{\alpha_1}} = \theta_1(e_{11})$.

$$\begin{aligned}
\sigma_A(s)_{|\alpha_1} &= \sigma_{A_n}(...(\sigma_{A_1}(s)))_{|\alpha_1} \\
&= \sigma_{A_n}(...(\sigma_{A_1}(s)_{|\alpha_1})) \\
&= \sigma_{A_n}(...(\sigma_{A_1}(\sigma_{A_1}(s)_{|\alpha_1}))) \quad \text{car } \sigma_{A_1} \text{ est idempotente} \\
&= \sigma_{A_n}(...(\sigma_{A_1}(\theta_1(e_{11})))). \\
&= \sigma_A(\theta_1(e_{11}))
\end{aligned}$$

Quand une équation est appliquée, elle ne doit contenir que des nouvelles variables : soit (e'_{11}, e'_{21}) une variante de l'équation (e_{11}, e_{21}), avec des nouvelles variables, soit σ'_1 une substitution de renommage telle que $e'_{11} = \sigma'^{-1}_1(e_{11})$ et $e'_{21} = \sigma'^{-1}_1(e_{21})$.
La présence de cette substitution σ'_1 ne se justifie que pour que l'équation soit appliquée avec des nouvelles variables.
L'équation (e'_{11}, e'_{21}) peut donc être appliquée sur $\sigma_A(s)$, à la position α_1 et avec la substitution $\sigma'_1 \circ \theta_1 \circ \sigma_A$:

$$\lambda.\sigma_A(s) \xrightarrow{eq(\alpha_1, \sigma'_1 \circ \theta_1 \circ \sigma_A, e'_{11}, e'_{21})} \lambda.s'_1.$$

Nous avons alors :

$$\begin{aligned}
s'_1 &= \sigma_A(s)\ [\alpha_1 \leftarrow \sigma_A(\theta_1(\sigma'_1(e'_{21})))] \\
&= \sigma_A(s)\ [\alpha_1 \leftarrow \sigma_A(\theta_1(e_{21}))]
\end{aligned}$$

et

$$\begin{aligned}
\sigma_A(s_1) &= \sigma_A(\sigma_{A_1}(s)\ [\alpha_1 \leftarrow \theta_1(e_{21})]) \\
&= \sigma_A(\sigma_{A_1}(s))\ [\alpha_1 \leftarrow \sigma_A(\theta_1(e_{21}))] \\
&= \sigma_A(s)\ [\alpha_1 \leftarrow \sigma_A(\theta_1(e_{21}))] \qquad \text{car } \sigma_{A_1} \circ \sigma_A = \sigma_{A_1} \circ \sigma_{A_1} \circ ... \circ \sigma_{A_n} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{et } \sigma_{A_1} \text{ idempotente} \\
&= s'_1
\end{aligned}$$

(b) Traitement de la deuxième équation

Comme la transformation $eq(\alpha_2, \theta_2, e_{12}, e_{22})$ est applicable : $\sigma_{A_2}(s_1)_{|\alpha_2} = \theta_2(e_{12})$.
Comme les variables qui disparaissent ne réapparaissent pas :
$Var(s_1) \cap Dom(\sigma_{A_1}) = \emptyset$ donc $\sigma_{A_1}(s_1) = s_1$.
De même $Var(\theta_2(e_{12})) \cap Dom(\sigma_{A_1}) = \emptyset$ donc $\theta_2(e_{12}) = \sigma_{A_1}(\theta_2(e_{12}))$.

$$\begin{aligned}
\sigma_A(s_1)_{|\alpha_2} &= \sigma_{A_n}(...(\sigma_{A_2}(\sigma_{A_1}(s_1))))_{|\alpha_2} \\
&= \sigma_{A_n}(...(\sigma_{A_2}(s_1)))_{|\alpha_2} \qquad \text{car } \sigma_{A_1}(s_1) = s_1 \\
&= \sigma_{A_n}(...(\sigma_{A_2}(s_1)_{|\alpha_2})) \\
&= \sigma_{A_n}(...(\sigma_{A_2}(\sigma_{A_2}(s_1)_{|\alpha_2}))) \qquad \text{car } \sigma_{A_2} \text{ est idempotente} \\
&= \sigma_{A_n}(...(\sigma_{A_2}(\theta_2(e_{12})))) \\
&= \sigma_{A_n}(...(\sigma_{A_2}(\sigma_{A_1}(\theta_2(e_{12})))))) \qquad \text{car } \theta_2(e_{12}) = \sigma_{A_1}(\theta_2(e_{12})) \\
&= \sigma_A(e_{12})
\end{aligned}$$

Nous réitérons le même principe que dans le cas précédent pour engendrer des nouvelles variables pour l'équation appliquée. Nous avons alors :

$$\lambda.\sigma_A(s_1) \xrightarrow{eq(\alpha_2, \sigma'_2 \circ \theta_2 \circ \sigma_A, e'_{21}, e'_{22})} \lambda.s'_2.$$

Nous avons $Var(\sigma_{A_2}(s_1)) \cap Dom(\sigma_{A_1}) = \emptyset$, toujours car les variables qui disparaissent ne peuvent pas réapparaître, donc $\sigma_{A_1}(\sigma_{A_2}(s_1)) = \sigma_{A_2}(s_1) = \sigma_{A_2}(\sigma_{A_1}(s_1))$.
Nous avons alors :

$$
\begin{array}{lll}
s'_2 & = \sigma_A(s_1)[\alpha_2 \leftarrow \sigma_A(\theta_2(\sigma'_2(e'_{22})))] & \\
 & = \sigma_A(s_1)[\alpha_2 \leftarrow \sigma_A(\theta_2(e_{22}))] & \\
\sigma_A(s_2) & = \sigma_A(\sigma_{A_2}(s_1)[\alpha_2 \leftarrow \theta_2(e_{22})]) & \\
 & = \sigma_A(\sigma_{A_2}(s_1))[\alpha_2 \leftarrow \sigma_A(\theta_2(e_{22}))] & \\
 & = \sigma_{A_n}(...\sigma_{A_2}(\sigma_{A_1}(\sigma_{A_2}(s_1))))[\alpha_2 \leftarrow \sigma_A(\theta_2(e_{22}))] & \\
 & = \sigma_{A_n}(...\sigma_{A_2}(\sigma_{A_2}(\sigma_{A_1}(s_1))))[\alpha_2 \leftarrow \sigma_A(\theta_2(e_{22}))] & \\
 & = \sigma_A(s_1)[\alpha_2 \leftarrow \sigma_A(\theta_2(e_{22}))] & \text{car } \sigma_{A_2} \text{ est idempotente} \\
 & = s'_2 &
\end{array}
$$

(c) Nous réitérons le même raisonnement sur les équations suivantes.

Nous obtenons alors :

$$
\begin{array}{ll}
 & \lambda Var(\sigma_A(s)).\sigma_A(s) \\
\xrightarrow{eq(\alpha_1,\sigma'_1\circ\theta_1\circ\sigma_A,e'_{11},e'_{21})} & \lambda Var(\sigma_A(s_1)).\sigma_A(s_1) \\
\xrightarrow{eq(\alpha_2,\sigma'_2\circ\theta_2\circ\sigma_A,e'_{12},e'_{22})} & \lambda Var(\sigma_A(s_2)).\sigma_A(s_2) \\
 & ... \\
\xrightarrow{eq(\alpha_{n-1},\sigma'_{n-1}\circ\theta_{n-1}\circ\sigma_A,e'_{1n-1},e'_{2n-1})} & \lambda Var(\sigma_A(s_{n-1})).\sigma_A(s_{n-1})
\end{array}
$$

Donc $\lambda Var(\sigma_A(s)).\sigma_A(s) \xrightarrow{A_E} \lambda Var(\sigma_A(s_{n-1})).\sigma_A(s_{n-1})$.

3. Nous venons de vérifier deux des trois conditions d'application de eq :
   – égalité des sous-termes ;
   – et nouvelle variante de l'équation.

   Montrons maintenant que la condition sur l'image de $\sigma'_j \circ \theta_j \circ \sigma_A$ est vérifiée. C'est cette condition qui assure que l'application de l'équation n'introduit que des nouvelles variables.

   Pour cela, nous allons d'abord montrer que, cette suite d'équations étant issue d'une suite de transformations, toute variable qui disparaît ne peut pas être réintroduite.

   Soit x une variable, si x disparaît alors :
   $\exists j$ tel que $x \in Var(\sigma_A(s_{j-1}))$ et $x \notin Var(\sigma_A(s_j))$.
   Donc x n'apparaît que dans $\sigma_A(s_{j-1})_{|_{\alpha_j}} = \sigma_A(\theta_j(e_{1j}))$ et $x \notin \sigma_A(\theta_j(e_{2j}))$.

   Si $x \in Dom(\sigma_A)$, étant donné que $Dom(\sigma_A) \cap Im(\sigma_A) = \emptyset$, alors $x \notin Im(\sigma_A)$. Ceci est en contradiction avec $x \in Var(\sigma_A(s_{j-1}))$.

   Donc $x \notin Dom(\sigma_A)$. x n'apparaît donc que dans $s_{j-1_{|_{\alpha_j}}}$.

   Après application de $eq(\alpha_j, \theta_j, e_{1j}, e_{2j})$, x va disparaître de la transformation initiale. x ne pouvant pas réapparaître (propriété 2.5.2, page 51), $x \notin Im(\sigma_{A_j} \circ ... \circ \sigma_{A_n})$.

   Si $x \in Im(\sigma_{A_1} \circ ... \circ \sigma_{A_{j-1}})$, les variables avec lesquelles x est liée (c'est-à-dire celles qui pourraient faire apparaître x par application de σ_A) n'apparaissent pas dans le nouveau problème, car elles appartiennent à $Dom(\sigma_A)$ et n'appartiennent donc pas à $Im(\sigma_A)$. Elles disparaissent donc par application de σ_A .

   Donc x ne pourra pas réapparaître dans le nouveau problème.

   Toutes les variables qui vont potentiellement apparaître par application d'une équation n'ont pas pu disparaître précédemment, donc soit elles appartiennent au terme sur lequel est appliquée la transformation, soit ce sont de nouvelles variables. Notre condition sur les images des substitutions $\sigma'_j \circ \theta_j \circ \sigma_A$ est donc vérifiée.

4. À chaque étape, les variables du domaine de σ_{A_i} sont éliminées. En effet, elles ne seront plus présentes dans les termes suivants, puisque la substitution est appliquée et d'après le lemme précédent $Dom(\sigma_{A_i}) \cap Im(\sigma_{A_i}) = \emptyset$.
   Ainsi : $\forall i \leq n-1,\ Var(s_{n-1}) \cap Dom(\sigma_{A_i}) = \emptyset$, donc $\forall i \leq n-1, \sigma_{A_i}(s_{n-1}) = s_{n-1}$.
   $\sigma_A(s_{n-1})$=$\sigma_{A_n}(...(\sigma_{A_1}(s_{n-1})))$
   =$\sigma_{A_n}(...(\sigma_{A_2}(s_{n-1})))$
   =...
   =$\sigma_{A_n}(s_{n-1})$
   =r

   Nous avons vu que $\lambda Var(\sigma_A(s)).\sigma_A(s) \xrightarrow{A_E} \lambda Var(\sigma_A(s_{n-1})).\sigma_A(s_{n-1})$,
nous avons donc $\lambda Var(\sigma_A(s)).\sigma_A(s) \xrightarrow{A_E} \lambda Var(r).r$.

□

Corollaire 2.5.2
Pour tous les termes r et s et toute suite de transformations A tels que $\lambda Var(s).s \xrightarrow{A} \lambda Var(r).r$, si nous notons $\sigma = \sigma_{A|Var(s)}$ (restriction de σ_A à $Var(s)$), nous avons :

$$\lambda Var(s).s \xrightarrow{A_\sigma} \lambda Var(\sigma_A(s)).\sigma_A(s) \xrightarrow{A_E} \lambda Var(r).r.$$

Preuve L'application de $\xrightarrow{A_\sigma}$ va appliquer σ_A et la suite a été prouvée par le lemme précédent.

Montrons que toutes les transformations de A_σ peuvent être appliquées :

- Le domaine de σ étant $Var(s)$, toutes les transformations $cste$ pourront être appliquées.
- Nous voulons appliquer une transformation $lie(x, y)$. Soit y est une nouvelle variable et nous pouvons l'appliquer. Soit elle ne l'est pas et elle est apparue précédemment dans les termes. Comme elle appartient à $Im(\sigma)$. Elle n'appartient pas à $Dom(\sigma)$. Elle n'a donc pas pu disparaître, elle appartient donc au terme sur lequel est appliquée la transformation.
- Nous voulons appliquer une transformation $comp(x, t)$. $Var(t)$ ne possède pas forcément que des nouvelles variables.
  Les variables qui posent problème sont les variables qui ne sont pas nouvelles dans t. Nous allons définir σ qui associe à ces variables une nouvelle variable. Nous avons alors $\sigma(t) = t'$ et $t = \sigma^{-1}(t')$. Nous pouvons alors appliquer $comp(x, t')$.
  Pour obtenir le même effet que l'application de $comp(x, t)$, pour toutes les variables v de $Dom(\sigma)$, nous allons ajouter la transformation $lie(\sigma^{-1}(v), v)$. $\sigma^{-1}(v)$ apparaît dans le terme puisque nous venons d'appliquer $comp(x, t')$. v est une variable qui est déjà apparue dans les termes (puisque ce n'est pas une nouvelle variable). Étant donné qu'elle appartient à l'image de σ, elle ne peut pas être dans son domaine et n'a donc pas disparu. Donc v appartient au terme sur lequel nous voulons appliquer lie.
  Exemple :
  $\lambda x\lambda y.x*y \xrightarrow{comp(y,t_1+t_2)} \lambda x\lambda t_1\lambda t_2.x*(t_1+t_2) \xrightarrow{lie(t_1,x)} \lambda x\lambda t_2.x*(x+t_2) \xrightarrow{lie(t_2,x)} \lambda x.x*(x+x)$
  La substitution associée est $\{y \mapsto x+x, t_1 \mapsto x, t_2 \mapsto x\}$, nous souhaiterions donc appliquer $\xrightarrow{comp(y,x+x)}$ sur $\lambda x\lambda y.x*y$. Or, x n'est pas une nouvelle variable,

nous allons donc, dans un premier temps, substituer un terme composé de nouvelles variables, avant de lier ces variables à x. Pour cela, nous ferons :
$\lambda x \lambda y.x * y \xrightarrow{comp(y,t_3+t_3)} \lambda x \lambda t_3.x * (t_3 + t_3) \xrightarrow{lie(t_3,x)} \lambda x.x * (x + x)$.

Ceci est un artifice pour que les conditions d'applications des transformations soient valides et que nous puissions utiliser les résultats (idempotence des substitutions engendrées, ré-ordonnancement, ...) prouvés sur ces transformations.

□

Conservation de A_E par application de σ

Propriété 2.5.4
Soient s et r deux termes et A_E une suite de transformations « eq » tels que
$\lambda Var(s).s \xrightarrow{A_E} \lambda Var(r).r$.
Alors, pour toute substitution σ, $\lambda Var(\sigma(s)).\sigma(s) \xrightarrow{A_E} \lambda Var(\sigma(r)).\sigma(r)$.

Preuve Nous allons faire une preuve par récurrence sur la taille de A_E.

Cas d'arrêt :
Si A_E est de taille 0, la propriété est vraie.

Poursuite :
Supposons que la propriété est vraie pour des transformations de taille $n-1$, montrons qu'elle est vraie pour une transformation de taille n.

Nous avons :
- $A_E = eq(\alpha, \theta, e_1, e_2); A_1$
- $\lambda Var(s).s \xrightarrow{eq(\alpha,\theta,e_1,e_2)} \lambda Var(a).a \xrightarrow{A_1} \lambda Var(r).r$
- $s_{|\alpha} = \theta(e_1)$
- $\theta(e_2) = a_{|\alpha}$

Donc $\sigma(s_{|\alpha}) = \sigma(\theta(e_1))$ et $\sigma(\theta(e_2)) = \sigma(a_{|\alpha})$.

Nous avons donc :
$\lambda Var(\sigma(s)).\sigma(s) \xrightarrow{eq(\alpha,\theta\circ\sigma,e_1,e_2)} \lambda Var(\sigma(a)).\sigma(a) \xrightarrow{A_1} \lambda Var(\sigma(r)).\sigma(r)$.

Au besoin, nous appliquons les mêmes « artifices » que précédemment pour introduire une équation avec des nouvelles variables si les variables de e_1 et e_2 font partie de l'image de σ. □

Nous avons rapidement rappelé les éléments essentiels sur les spécifications algébriques que nous allons exploiter pour exprimer les domaines applicatifs et les services disponibles. Nous avons ensuite défini l'ensemble des services qui peuvent être rendus dans un domaine à partir des services disponibles. Nous allons maintenant étudier un premier algorithme de recherche issu de la similitude entre l'unification équationnelle et la comparaison entre un service et la requête de l'utilisateur.

Chapitre 3

Le courtage basé sur le filtrage équationnel

Selon notre définition des services répondant à la requête, le courtage doit construire un sous-ensemble des services s et des substitutions σ tels que $\sigma(s) =_{\mathcal{E}} r$. Il s'agit d'un problème de filtrage équationnel. Dans notre cas, la requête ne possédant que des constantes, ce problème est équivalent à un problème d'unification équationnelle [Bür89].

Nous proposons un premier algorithme, basé sur les travaux de Gallier et Snyder [GS89] sur l'unification équationnelle. Un des avantages de cette approche était la possibilité de réutiliser leur travail pour montrer que l'algorithme était correct et complet. Nous avons pu utiliser la preuve de la correction pour prouver la correction de notre algorithme. La complétude n'a pas pu être montrée, car notre problème n'étant pas exactement le même que le leur, nous avons adapté leur système et n'avons pas pu adapter simplement leur preuve à notre problème particulier.

Cet algorithme possède également des propriétés intéressantes sur la séparation des sous-problèmes. En effet, tous les sous-problèmes issus du problème initial sont traités de façon totalement indépendante. Cela permet de le paralléliser et d'envisager une réutilisation plus efficace du travail déjà réalisé.

3.1 Le système d'inférence de Gallier et Snyder

À notre connaissance, aucun algorithme n'avait été proposé pour résoudre le problème du filtrage équationnel dans le cadre d'une théorie quelconque. Nous sommes donc partis de travaux réalisés dans le cas de l'unification équationnelle. Notre requête n'ayant que des constantes, les problèmes d'unification équationnelle et de filtrage équationnel sont équivalents. De plus, au cours de traitement des problèmes de filtrage équationnel, nous pouvons être amenés à traiter des problèmes d'unification équationnelle (comme nous allons le voir dans la suite), cela justifie d'autant plus le fait de s'appuyer sur l'unification équationnelle.

Dans [GS89], Gallier et Snyder proposent des systèmes de déduction pour résoudre les problèmes d'unification équationnelle généraux. Ils proposent deux systèmes de déduction et prouvent leur complétude (le système est complet si et seulement si, pour tout ensemble $\mathcal{E}$ d'équations, un ensemble complet d'E-unificateurs peut être énuméré en utilisant ses transformations). Ils prouvent aussi leur correction (toute substitution trouvée est bien un E-unificateur). Nous nous sommes principalement intéressés au premier système. Le second a été proposé pour pallier le problème de non déterminisme dans les séquences d'application

d'une règle (Root Imitation). Nous avons réglé ce problème, d'une autre façon, en exploitant la structure particulière de notre problème. Ce système est composé de cinq règles :

1. **Trivial** :
$$\{\langle u, u\rangle\} \cup S \Rightarrow S$$
2. **Term Decomposition** :
Pour tout f $\in \Sigma_n$, pour un $n > 0$,
$$\{\langle f(u_1, ..., u_n), f(v_1, ..., v_n)\rangle\} \cup S \Rightarrow \{\langle u_1, v_1\rangle, ..., \langle u_n, v_n\rangle\} \cup S$$
3. **Variable Elimination** :
$$\{\langle x, v\rangle\} \cup S \Rightarrow \{\langle x, v\rangle\} \cup \sigma(S)$$
où $\langle x, v\rangle$ n'est pas une paire résolue dans S telle que $x \notin Var(v)$ et $\sigma = [v/x]$.
4. **Root Rewriting** :
Soient u et v deux termes, si u ou v n'est pas une variable, supposons qu'il s'agisse de u, alors :
$$\{\langle u, v\rangle\} \cup S \Rightarrow \{\langle u, l\rangle, \langle r, v\rangle\} \cup S$$
où $l = r$ est une variante d'une équation telle que $Var(l, r) \cap (Var(S) \cup Var(u, v)) = \emptyset$ et si ni u ni l n'est une variable, alors $Root(u) = Root(l)$.
Root Rewriting ne doit pas être appliquée sur la paire $\langle u, v\rangle$.
5. **Root Imitation** :
Si x est une variable et $f \in \Sigma_n$ avec $n > 0$, alors nous avons :
$$\{\langle x, v\rangle\} \cup S \Rightarrow \{\langle x, f(y_1, ...y_n)\rangle, \langle x, v\rangle\} \cup S$$
où $y_1, ..., y_n$ sont des $nouvelles$ variables et si v n'est pas une variable, alors $f = Root(v)$. Nous appliquons immédiatement Variable Elimination sur la nouvelle paire $\langle x, f(y_1, ..., y_n)\rangle$.

3.2 Autres travaux sur l'unification équationnelle

Dans [GS89] Gallier et Snyder proposent deux systèmes de déduction. Le second a été défini pour améliorer les performances du premier, en utilisant une variante de la règle de para-modulation. D'autres travaux ont par la suite été proposés pour améliorer ce second système. Nous pouvons par exemple citer [DJ90, SA94].

Nous avons néanmoins choisi de conserver le premier système proposé par Gallier et Snyder comme base de notre algorithme. L'approche de celui-ci consiste à rendre les symboles à la racine égaux et, seulement dans le cas où cela est possible, de s'intéresser aux sous-termes. Les autres systèmes permettent de travailler sur les sous-termes quel que soit leur position. Ce choix a été fait pour plusieurs raisons.

D'abord, Gallier et Snyder ont introduit ce second système pour palier au fait que la règle « Root Imitation » peut être appliquée indéfiniment. Cependant, cette règle n'est nécessaire que dans le cas où les deux termes que nous voulons unifier ont des variables communes. Rappelons que dans notre cas les deux termes du problème initial ont des ensembles de variables disjoints (puisque l'un des termes n'est composé que de constantes). Cette propriété étant conservée sur la majorité des sous-problèmes engendrés, cette règle n'est appliquée que dans certains cas très particuliers, que nous détaillerons plus particulièrement dans la suite.

Ces améliorations permettent, dans le cadre général de l'unification équationnelle, de reduire le nombre de règles d'inférence appliquées et le nombre de sous-problèmes engendrés, mais pas forcément le temps de calcul. En effet, même si certaines heuristiques permettent de réduire le nombre de sous-problèmes engendrés, le temps de traitement de chaque sous-problème pouvant être plus important, nous ne gagnons pas obligatoirement en terme de temps d'exécution. Les techniques basées sur la para-modulation impliquant une recherche des équations et des positions où elles peuvent être appliquées, le traitement d'un sous-problème sera plus coûteux. En effet, il faudra étudier tous les sous-termes et plus seulement regarder le symbole à la racine. Cela laisse supposer que la réduction du nombre de sous-problèmes ne compensera pas le travail supplémentaire nécessaire au traitement de chaque sous-problème.

D'autres stratégies, comme la stratégie basique, ont été développées pour limiter les positions où les règles d'inférences sont applicables et donc limiter le coût de traitement d'un sous-problème. Cette stratégie a été initialement proposée par Hullot [Hul80] dans le cadre du narrowing. Elle consiste à ne pas appliquer de règles d'inférence à des positions introduites par des substitutions, c'est-à-dire à des positions où il y avait une variable qu'une substitution a remplacée par un terme. Ce terme ne pourra alors pas être modifié directement par application d'une règle d'inférence. Cela permet de limiter le nombre de sous-problèmes engendrés en limitant les positions où peuvent être appliquée la règle.

Cela n'est pas utilisable dans le cadre de notre choix de n'appliquer les équations qu'au niveau de la racine des termes présents dans les sous-problèmes.

D'autres optimisations ont été introduites pour traiter le cas des systèmes de réécriture. Nous pouvons notamment citer les travaux de Michael Hanus [Han94, AEH94]. Une de nos contraintes de départ était de rester dans le cadre général, ces techniques ne pouvaient donc pas servir de base à nos algorithmes. Néanmoins, il serait intéressant d'étudier ce qui pourrait être réalisé en introduisant des simplifications lors de la résolution des problèmes. Ces simplifications pourraient être réalisées à partir d'un système de réécriture, construit à partir d'un sous-ensemble de l'ensemble des équations décrivant le domaine. Le problème du choix de ce sous-ensemble, qui pour répondre à nos contraintes devrait être automatisé, se poserait alors. Nous verrons dans la suite, que nous avons introduit cette notion de simplification (en option) dans le cas très précis de la relance de l'algorithme pour la composition. Ceci est développé dans la sous-section 3.5.4, page 75. Le choix des équations de simplification est fait par le spécialiste du domaine. Il faudrait modifier cela pour le faire de façon automatique.

3.3 L'algorithme basé sur le filtrage équationnel

À partir du système de Gallier et Snyder, nous avons réalisé une première version non typée de l'algorithme.

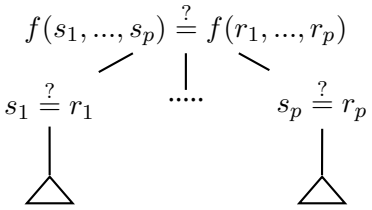
3.3.1 Le principe de base

Le principe de l'algorithme de Gallier et Snyder est d'appliquer les règles de transformations dans un ordre quelconque et tant que cela est possible. Il n'était évidemment pas envisageable de garder ce fonctionnement pour notre algorithme, car il est indéterministe et peut boucler avant d'avoir engendré toutes les solutions finies. Nous ne trouverions donc pas nécessairement les solutions finies en un temps fini. Dans l'optique d'un algorithme « exploitable », il fallait des garanties sur la terminaison de l'algorithme (nous introduisons

la notion de « quantité d'énergie », qui, si elle est finie, entraîne la terminaison de l'algorithme). De plus, l'ensemble des solutions acceptables ne sont pas les mêmes. Pour qu'une solution ait un sens dans notre cas, il faut que les variables du service soient assignées à une constante (du domaine ou de la requête), à un terme ne contenant que des constantes ou des variables libres (terme qui doit pouvoir être rendu à l'aide des services disponibles) ou à une variable libre (c'est-à-dire qu'aucune contrainte particulière n'est imposée à la variable, elle peut prendre une forme quelconque). Ces variables libres proviennent de l'application d'une équation dans le cas où les mêmes variables ne sont pas présentes dans les deux membres (voir définition 2.1.15, page 34).

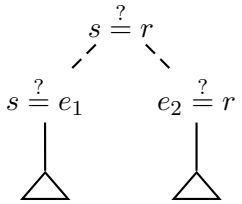
L'algorithme de base est très simple. Une solution est obtenue par une suite de décompositions de termes et d'applications d'équations. Quand une de ces transformations est applicable sur un problème, l'algorithme l'applique. Si plusieurs transformations sont applicables, l'algorithme construit les différents arbres engendrés par l'application de toutes les transformations possibles. Cela conduit, en cas de réussite, à des solutions différentes. Les sous-problèmes obtenus sont traités de façon indépendante : les résultats obtenus lors du traitement d'un fils ne sont pas utilisés lors du traitement des fils suivants. Si les contraintes sur les variables obtenues dans chaque sous-problème sont compatibles, elles sont alors combinées pour construire la solution finale. Sinon il y a échec de la comparaison.

Décomposition Quand les symboles à la racine des deux termes sont égaux, une transformation applicable est une décomposition de ces termes pour comparer deux à deux les fils.



Application d'équations À tout moment (cette condition sera modifiée par la suite en tenant compte de la quantité d'énergie et des contraintes sur la forme de l'équation) une équation peut être appliquée.

Application de l'équation (e_1, e_2) :



Quand une équation est appliquée, c'est une version de l'équation avec des nouvelles variables qui est choisie. Les variables introduites par les équations sont appelées « variables intermédiaires » et notées x_i. Ce sont elles qui relient les sous-problèmes entre eux.

Ces deux transformations sont appliquées jusqu'à obtenir des formes résolues.

Les formes résolues Les formes résolues sont les feuilles $s \stackrel{?}{=} r$ telles que :

- s et r sont deux constantes identiques ;
- s est une variable ;
- ou r est une variable intermédiaire.

Si, dans une branche, il n'est pas possible d'atteindre une de ces formes résolues, la comparaison échoue.

Suppression des variables intermédiaires Nous construisons la solution à partir des formes résolues. Les équations introduisent des variables qui n'ont pas de sens dans notre problème, il faudra donc supprimer ces variables. Un graphe est créé à partir des dépendances obtenues en analysant la forme résolue. Puis, par parcours de ce graphe, les variables intermédiaires sont supprimées et les problèmes qui nécessitent un traitement supplémentaire sont relancés.

Nous construisons un graphe orienté de la façon suivante. Quand une forme résolue est :

- $s \stackrel{?}{=} r$, où s et r sont deux constantes identiques, nous n'ajoutons rien à notre graphe ;
- $s \stackrel{?}{=} r$, où s est une variable, nous ajoutons les nœuds s et r, s'ils ne sont pas déjà présents et l'arête $s \rightarrow r$ au graphe ;
- $s \stackrel{?}{=} r$, où r est une variable intermédiaire et s est une constante, nous ajoutons les nœuds s et r, s'ils ne sont pas déjà présents et l'arête $r \rightarrow s$ au graphe ;
- $s \stackrel{?}{=} r$, où r est une variable intermédiaire et s est une variable ou un terme, nous ajoutons les nœuds s et r, s'ils ne sont pas déjà présents et l'arête $s \rightarrow r$ au graphe.

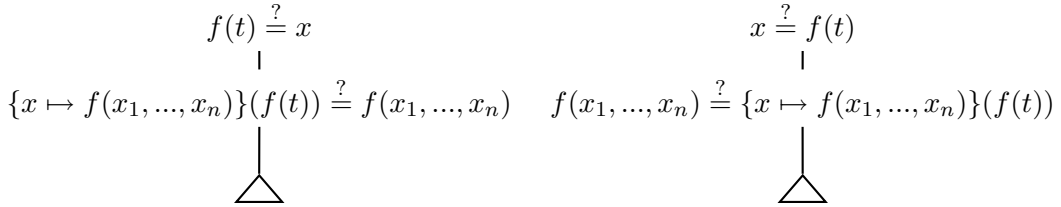
Nous obtenons ainsi un graphe dont les entrées sont des variables du service ou des termes et les sorties sont des termes, des constantes ou des variables libres.

Relances Lors du parcours du graphe, nous pouvons être amenés à relancer l'algorithme sur différents problèmes :

- Dans le cas où une entrée du graphe est un terme et une sortie associée est un terme, il faut relancer l'algorithme sur ce nouveau problème.
- Nous vérifions également qu'il n'y a pas deux valeurs possibles pour une même variable (deux sorties différentes pour la même entrée). Sinon il faut relancer le problème pour vérifier si elles sont égales modulo $\mathcal{E}$. Ce test s'effectue en utilisant notre algorithme, avec la quantité d'énergie restante et en vérifiant qu'une des solutions obtenues est l'identité. Néanmoins, lors de ces relances, nous pouvons être amenés à comparer deux termes contenant des variables libres. Nous n'avons alors plus un problème de filtrage équationnel, mais un problème d'unification équationnelle.

Des exemples de constructions de graphes et de relances sont donnés dans la partie 3.4 (page 66).

Introduction de symboles Comme nous venons de le voir, ce second type de relances engendre des problèmes d'unification équationnelle. Pour pouvoir les traiter correctement et tenir compte du fait que les variables libres peuvent prendre une forme quelconque, nous ajoutons les deux décompositions suivantes, où $f(t)$ est un terme et x une variable libre :



Cette transformation n'est utilisée que dans le cas très précis des relances et que lorsque x est une variable libre. Les variables x_i sont des nouvelles variables, elles mêmes considérées comme des variables libres.

Une fois tous les problèmes relancés, la suppression des variables intermédiaires produit une liste d'affectations (variable du service / terme, constante de la requête ou du domaine ou variable libre). Cette solution est celle qui nous intéresse. Dans le cas où il y a une affectation à un terme, le problème global est relancé avec ce terme comme requête, car celui-ci nécessite au moins une composition.

3.3.2 Preuve de la correction

Notre algorithme consiste à appliquer les règles de Gallier et Snyder dans un ordre particulier.

En effet, la décomposition de l'arbre est l'application de la règle **Term Decomposition** du système de Gallier et Snyder. À partir d'un nœud $f(u_1, ..., u_n) \stackrel{?}{=} f(v_1, ..., v_n)$, nous générons les sous-problèmes $\{u_1 \stackrel{?}{=} v_1$,...,$u_n \stackrel{?}{=} v_n\}$.

L'application d'une équation est l'application de la règle **Root Rewriting** du système de Gallier et Snyder. Comme dans leur règle, deux sous-problèmes sont engendrés et liés par les nouvelles variables de l'équation. Nous verrons dans la suite que nous avons les mêmes contraintes sur les symboles à la racine que celles liées à l'application de cette règle.

La suppression des variables intermédiaires s'effectue par propagation des valeurs associées à ces variables. Cela équivaut à appliquer la substitution qui permet de les éliminer. Il s'agit de l'application de la règle **Variables Elimination**. La différence est que, dans notre algorithme, les variables intermédiaires disparaissent car elles n'ont pas de sens dans le problème initial.

La transformation d'introduction d'un symbole pour les variables libres est l'application de la règle **Root Imitation**.

Une fois que les variables intermédiaires ont été supprimées, les formes résolues sont les mêmes que celles de Gallier et Snyder. Donc, si l'algorithme abouti à une de ces formes, il existera une suite de transformations dans le système de Gallier et Snyder conduisant à cette forme. La solution sera donc un unificateur équationnel de la requête et du service. Les contraintes spécifiques à notre problème (sur la forme des substitutions solutions) étant explicitement vérifiées par l'algorithme, celui-ci est correct.

3.3.3 Discussion sur la complétude

Nous souhaitions utiliser la preuve de complétude du système de Gallier et Snyder pour prouver cette même propriété sur notre algorithme. En effet, notre problème est un sous-problème du leur et notre algorithme implante une heuristique particulière d'ordre d'application de leurs règles.

La difficulté principale est liée à une définition non constructive des solutions que l'application des règles permet de construire (σ tel que $\sigma(s) =_{\mathcal{E}} \sigma(r)$). Il n'était ainsi pas simple d'adapter la notion de solution au sous-ensemble qui nous concerne et de montrer que notre ordre d'application des règles engendre bien toutes les solutions qui nous intéressent.

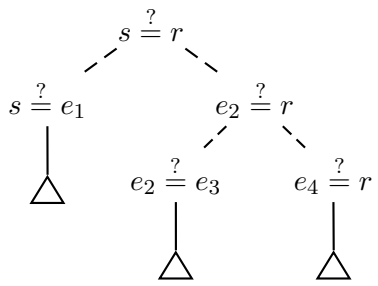
Nous n'avons pas réussi à conclure en un temps raisonnable que notre ordre particulier d'application des règles, pour la forme particulière de notre problème, n'entraînait pas la suppression de solutions. Pourtant, toutes leurs règles apparaissent dans notre algorithme, seul le moment où nous les appliquons change et certaines contraintes ont été durcies. Ces changements sont liés au fait que nos solutions ne sont qu'un sous-ensemble des leurs et que nos problèmes initiaux sont un sous-ensemble des problèmes possibles.

Néanmoins, nous avons fait une première étude pour déterminer l'impact de l'ordre dans lequel sont appliquées les transformations du système de Gallier et Snyder dans notre algorithme. Cette étude a été présentée dans [HP06] et complétée dans [HP05]. À l'aide d'un calcul sur les langages réguliers, qui représentent les traces d'applications des transformations, nous avons montré que tous les ordres étaient possibles, comme dans le cas du système de Gallier et Snyder. Le résultat découle de la simplification effectuée en ne considérant ni les contraintes d'applications des règles, ni les positions d'application des équations. L'objectif était de détecter d'éventuels oublis (mise au point de l'algorithme) et ne permettait pas d'assurer la complétude. nous n'avons ensuite pas poursuivi dans cette voie car la prise en compte de ces contraintes ne permettait plus de disposer de langages réguliers dont la comparaison était simple.

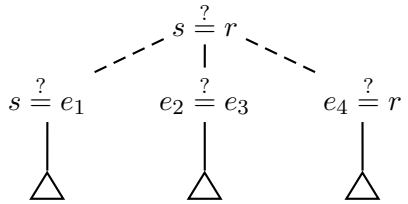
3.3.4 Traitement des équations

Dans l'algorithme que nous venons de présenter, une seule équation est appliquée sur un problème. Or, afin de réduire les possibilités dans le choix des équations, il est intéressant d'appliquer une suite d'équations. Nous avons ainsi la possibilité d'imposer des contraintes d'égalité sur les symboles à la racine des membres des équations de cette suite d'équations. Sur les sous-problèmes issus de l'application d'une transformation, une équation ne pourra pas être appliquée si une décomposition n'a pas été réalisée avant.

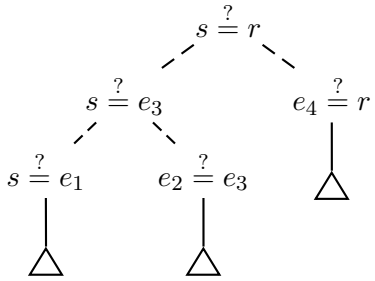
Un problème de la forme :



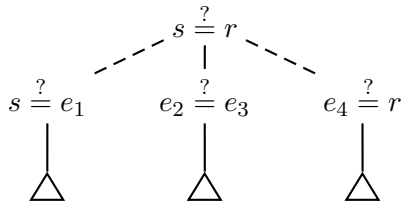
est en fait traité de la façon suivante :



Et un problème de la forme :



est également traité de la façon suivante :

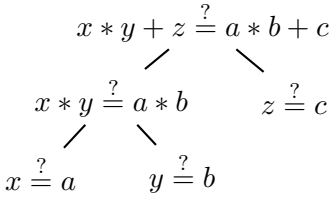


Il est évident que ces deux méthodes de traitement des équations sont équivalentes, puisque les sous-problèmes engendrés sont identiques.

3.4 Exemples

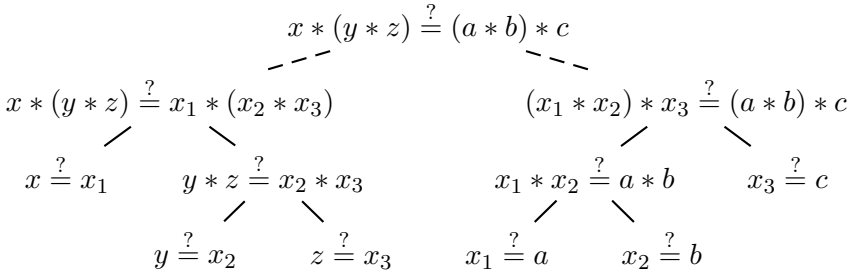
Dans cette section, nous allons illustrer le fonctionnement de l'algorithme sur des cas simples qui illustrent les points importants.

Décomposition Un service fourni est $s(x, y, z) = \lambda x \lambda y \lambda z.x * y + z$, la requête est $r = a * b + c$. Quelles que soient les équations, une solution est obtenue de la façon suivante :



Solution : $r = s(a, b, c)$.

Une équation, sans relance Un service fourni est $s(x, y, z) = \lambda x \lambda y \lambda z.x * (y * z)$, la requête est $r = (a * b) * c$. Parmi les équations se trouve $(v * u) * w = v * (u * w)$. Une solution est obtenue de la façon suivante :

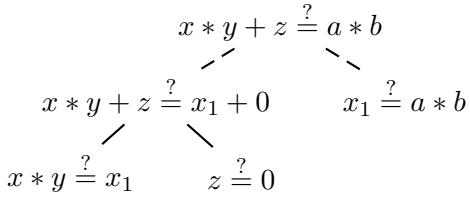


Les deux sous-problèmes $x * (y * z) \stackrel{?}{=} x_1 * (x_2 * x_3)$ et $(x_1 * x_2) * x_3 \stackrel{?}{=} (a * b) * c$ sont traités de façon totalement indépendante, sans tenir compte de l'existence du second problème. Il en est de même pour des branches issues d'une décomposition des termes, comme $x \stackrel{?}{=} x_1$ et $y * z \stackrel{?}{=} x_2 * x_3$. Ces résolutions peuvent être effectuées en parallèle.

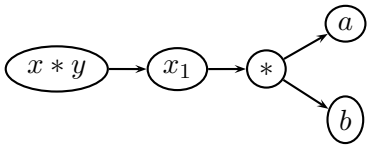
Les variables intermédiaires sont ensuite supprimées :

Solution : $r = s(a, b, c)$.

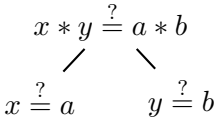
Une équation, avec relance Un service fourni est $s(x, y, z) = \lambda x \lambda y \lambda z.x * y + z$, la requête est $r = a * b$. Parmi les équations se trouve $v = v + 0$. Une solution est obtenue de la façon suivante :



Supprimons la variable intermédiaire :

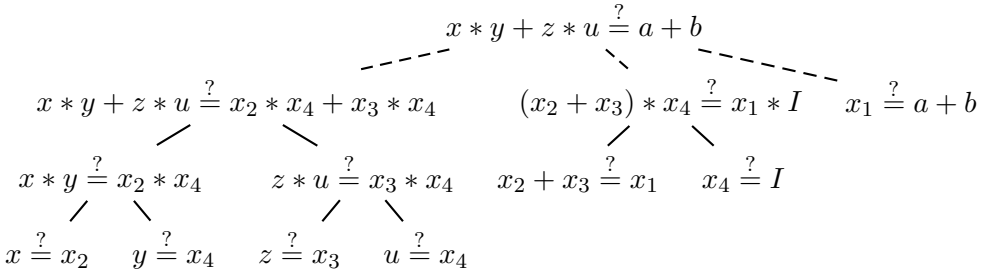


Relance sur le problème $x * y \stackrel{?}{=} a * b$:

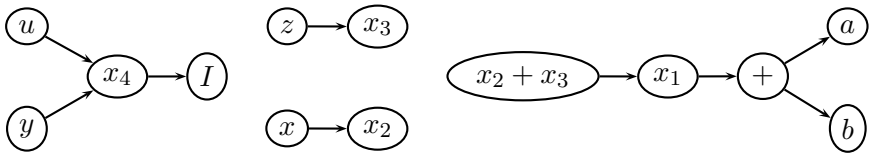


Solution : $r = s(a, b, 0)$.

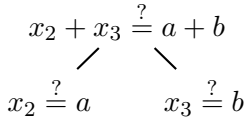
Transformation de taille 2 Un service fourni est $s(x, y, z, u) = \lambda x \lambda y \lambda z \lambda u.x * y + z * u$, la requête est $r = a + b$. Parmi les équations se trouvent $v = v * I$ et $(i + j) * k = i * k + j * k$. Une solution est obtenue de la façon suivante :



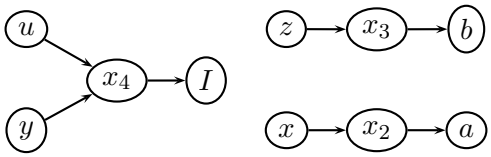
Suppression les variables intermédiaires :



Relance sur le problème $x_2 + x_3 \stackrel{?}{=} a + b$:

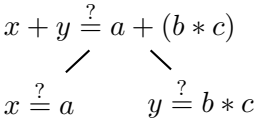


Nous complétons le graphe de suppression des variables intermédiaires et obtenons :

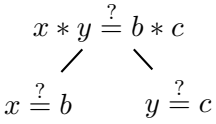


Solution : $r = s(a, I, b, I)$.

Composition Un service fourni est $add(x, y) = \lambda x \lambda y.x + y$ et un autre est $mult(x, y) = \lambda x \lambda y.x * y$, la requête est $r = a + (b * c)$. Quelles que soient les équations, une solution est obtenue de la façon suivante :

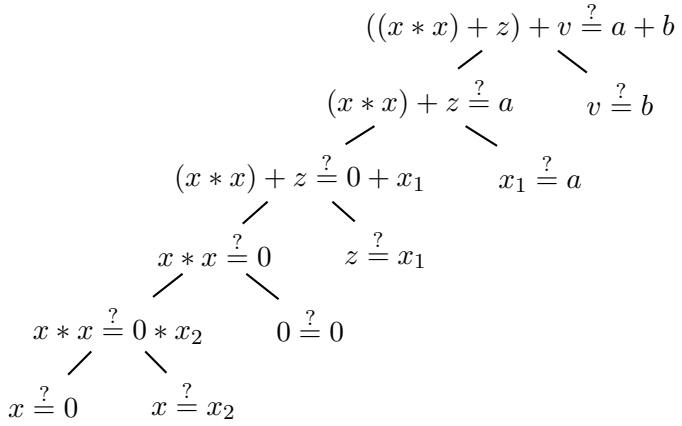


y est affecté à un terme, donc ce terme sera comparé avec tous les services disponibles et notamment avec $mult$:

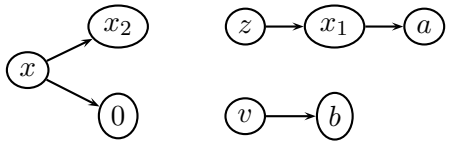


Solution : $r = add(a, mult(b, c))$.

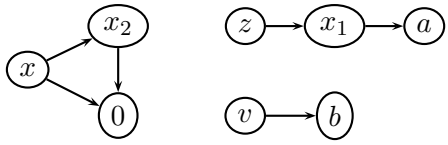
Des variables libres Un service fourni est $f(x, z, v) = \lambda x \lambda y \lambda z.((x * x) + z) + v$ et la requête est $a + b$. Parmi les équations se trouvent $0 + x_1 = x_1$ et $0 * x_2 = 0$. Une solution est obtenue de la façon suivante :



Suppression des variables intermédiaires :

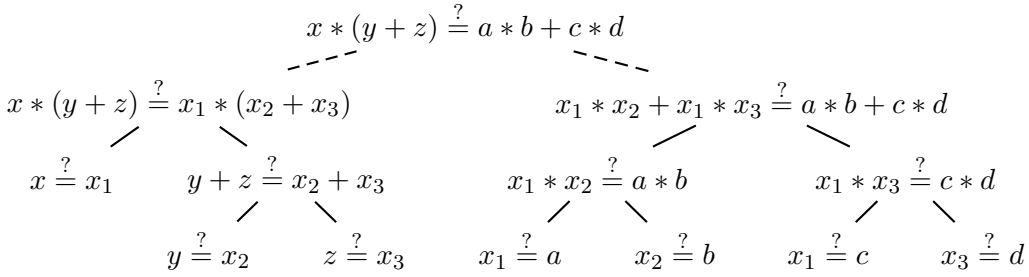


Relance sur le problème $x_2 \stackrel{?}{=} 0$ (second cas de relance, quand nous avons deux sorties différentes pour une même entrée). Ce problème est sous forme résolue. Nous complétons notre graphe de suppression des variables :

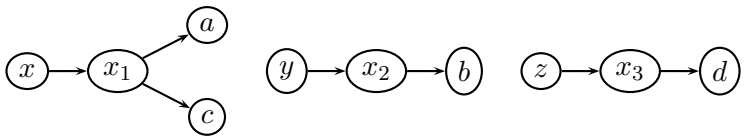


Solution : $r = f(0, a, b)$.

Échec Un service fourni est $s(x, y, z) = \lambda x \lambda y \lambda z.x*(y+z)$, la requête est $r = a*b+c*d$. Parmi les équations se trouve $i * (j + k) = i * j + i * k$. Une solution est recherchée de la façon suivante :



Suppression des variables intermédiaires :



x doit prendre deux valeurs différentes (a et c) : c'est un échec.

3.5 Réalisation

3.5.1 L'algorithme

L'algorithme naïf (Algorithme 1) a ensuite été amélioré pour être plus efficace.

Algorithme 1 Algorithme initial sans optimisation

$compare(s, r)$ =
compare s, r **avec**
| $Cste_1, Cste_2 \rightarrow$
si $Cste_1 \neq Cste_2$ **alors**
Échec
| $s, Var \rightarrow$
Assign(s,r)
| $Var, r \rightarrow$
Assign(s,r)
| $f(f_1, ..., f_n), f(g_1, ..., g_m) \rightarrow$
Comparer les fils f_i et g_i et appliquer les transformations qui conservent f
| $s, r \rightarrow$
Appliquer les transformations qui permettent de passer de la requête r au service s
fin compare

Il reste à définir ce que « Appliquer les transformations qui conservent f » et « Appliquer les transformations qui permettent de passer de la requête au service » signifient.

Optimisations Nous vérifions si la comparaison de tous les fils peut aboutir, c'est-à-dire s'il existe une transformation dont la taille en accord avec la quantité d'énergie restante, avant de faire le travail sur les fils.

Si nous effectuons un traitement en séquence des fils, nous prenons en compte la quantité d'énergie minimale pour obtenir une solution pour un fils et les fils suivants ne sont traités qu'avec l'énergie totale moins cette énergie. S'il n'y a pas de résultat sur un fils, le calcul n'est pas poursuivi.

Le même principe est appliqué pour les sous-problèmes obtenus à partir de l'application d'une transformation.

Ces deux optimisations ne sont plus applicables si nous faisons un traitement en parallèle des fils.

3.5.2 Traitement des équations

Nous avons vu que l'algorithme a été adapté pour appliquer une suite d'équations (transformation) et non une unique équation. Nous allons maintenant voir comment sont

construites les transformations et quelles sont les contraintes que nous leur imposons.

Une équation est composée de deux termes. Ces termes ont des symboles à la racine : n_1 et n_2. Nous noterons $n_1 \rightarrow n_2$ pour faire référence à l'une de ces équations. Nous noterons Nop (pour *No operator*) dans le cas des variables. Nous appellerons « transformation » toute suite d'équations.

– Dans un premier temps, les transformations qui permettent de passer du symbole à la racine r au symbole à la racine s étaient définies comme suit :

1. Les transformations de taille 1, applicables sur un problème dont le symbole à la racine de la requête est r et celui du service est s, sont les équations qui permettent de passer de r à s et celles qui introduisent s :
   $r \Rightarrow_1 s$

	équation 1	
r	r $\rightarrow$ s	s
r	Nop$\rightarrow$ s	s

2. Les transformations de taille 2, applicables sur un problème dont le symbole à la racine de la requête est r et celui du service est s sont, celles composées de deux équations et qui permettent de passer de r à s et celles qui introduisent s. Pour cela, nous passons par un symbole intermédiaire quelconque N_1 et construisons la transformation à partir de $r \Rightarrow_1 N_1$ et $N_1 \Rightarrow_1 s$:
   $r \Rightarrow_2 s \ = \ r \Rightarrow_1 N_1 \qquad N_1 \Rightarrow_1 s$

	équation 1	équation 2	
r	r $\rightarrow N_1$	$N_1 \rightarrow$ s	s
r	r $\rightarrow N_1$	Nop$\rightarrow$ s	s
r	Nop$\rightarrow N_1$	$N_1 \rightarrow$ s	s
r	Nop$\rightarrow N_1$	Nop$\rightarrow$ s	s

3. Les transformations de taille 3, applicables sur un problème dont le symbole à la racine de la requête est r et celui du service est s, sont celles composées de trois équations et qui permettent de passer de r à s et celles qui introduisent s. Pour cela, nous passons par deux symboles intermédiaires quelconques N_1 et N_2 et construisons la transformation à partir de $r \Rightarrow_1 N_1$, $N_1 \Rightarrow_1 N_2$ et $N_2 \Rightarrow_1 s$:
   $r \Rightarrow_3 s \ = \ r \Rightarrow_1 N_1 \qquad N_1 \Rightarrow_1 N_2 \qquad N_2 \Rightarrow_1 s$
   $r \Rightarrow_3 s \ = \ r \Rightarrow_1 N_1 \qquad N_1 \Rightarrow_2 s$
   $r \Rightarrow_3 s \ = \ r \Rightarrow_2 N_2 \qquad N_2 \Rightarrow_1 s$

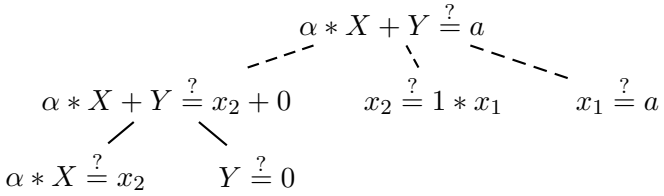
	équation 1	équation 2	équation 3	
r	r $\rightarrow N_1$	$N_1 \rightarrow N_2$	$N_2 \rightarrow$ s	s
r	r $\rightarrow N_1$	$N_1 \rightarrow N_2$	Nop$\rightarrow$ s	s
r	r $\rightarrow N_1$	Nop$\rightarrow N_2$	$N_2 \rightarrow$ s	s
r	r $\rightarrow N_1$	Nop$\rightarrow N_2$	Nop$\rightarrow$ s	s
r	Nop$\rightarrow N_1$	$N_1 \rightarrow N_2$	$N_2 \rightarrow$ s	s
r	Nop$\rightarrow N_1$	$N_1 \rightarrow N_2$	Nop$\rightarrow$ s	s
r	Nop$\rightarrow N_1$	Nop$\rightarrow N_2$	$N_2 \rightarrow$ s	s
r	Nop$\rightarrow N_1$	Nop$\rightarrow N_2$	Nop$\rightarrow$ s	s

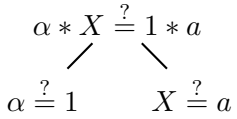
4. ...

Les transformations étaient donc de la forme :
$(e_{1,1}, e_{1,2}), ..., (e_{i,1}, e_{i,2}), ..., (e_{n,1}, e_{n,2})$ avec :
– $root(e_{1,1}) = root(r)$ ou $root(e_{1,1}) = Nop$,
– $root(e_{n,2}) = root(s)$,
– et $\forall i \in [1, n-1], root(e_{i,2}) = root(e_{i+1,1})$ ou $root(e_{i+1,1}) = Nop$.

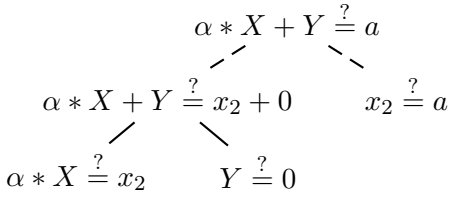
Notons ici la grande importance des « Nop » qui permettent d'introduire un symbole quelconque. Les éléments neutres jouent donc un rôle particulier.
– Dans un second temps, nous avons diminué cette grande importance des éléments neutres. En effet, alors que la présence du Nop se justifie en début de transformation, elle se justifie moins en milieu de transformation. En effet, elle est nécessaire pour introduire un symbole, mais en cours de transformation, elle n'est pas indispensable car cette équation peut être appliquée une fois la décomposition effectuée. Le symbole à la racine du nouveau problème est alors connu et les équations sont alors mieux choisies.

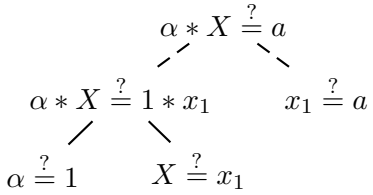
Exemple 3.5.1
La solution du problème $\alpha * X + Y \stackrel{?}{=} a$ *obtenue en appliquant la transformation* $(x_1, 1 * x_1), (x_2, x_2 + 0)$ *:*





est également obtenue de cette façon :





Les transformations sont donc de la forme :
$(e_{1,1}, e_{1,2}), ..., (e_{i,1}, e_{i,2}), ..., (e_{n,1}, e_{n,2})$ avec :
– $root(e_{1,1}) = root(r)$ ou $root(e_{1,1}) = Nop$,
– $root(e_{n,2}) = root(s)$,
– et $\forall i \in [1, n-1], root(e_{i,2}) = root(e_{i+1,1})$.

Les transformations qui introduisent un symbole sont celles qui permettent de passer de Nop à ce symbole et celles qui conservent un symbole sont celles qui permettent de passer de ce symbole à lui-même (diagonale de la matrice).

Les équations sont stockées dans une matrice, les indices de la case sont liés aux deux symboles à la racine des deux membres de l'équation. Une équation est donc stockée dans chaque sens d'orientation. Les transformations sont ensuite engendrées grâce à une fermeture transitive sur le nombre d'équations, en en supprimant certaines pour que les transformations respectent les contraintes énoncées ci-dessus. Il y a une matrice par taille de transformation (nombre d'équations composant la transformation).

Nous voyons apparaître ici une valeur importante : la taille des transformations. Pour un ensemble fini d'équations qui peuvent être appliquées, le nombre de transformations est, quant à lui, infini. Toutes les transformations ne peuvent donc pas être appliquées en un temps fini. Notre algorithme est donc paramétré par la taille maximale des transformations appliquées et le nombre total d'équations appliquées. Ces deux valeurs font partie des paramètres en entrée de l'algorithme. Ce sont des éléments de ce qui a été appelé précédemment « la quantité d'énergie » allouée à l'algorithme. L'autre valeur intervenant dans cette quantité d'énergie est la profondeur maximale de composition autorisée.

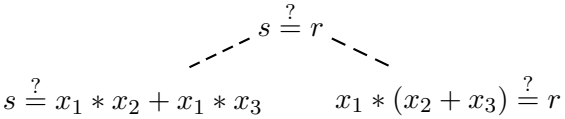
Les transformations choisies sont plus contraintes que celles du système de déduction de Gallier et Snyder. Elles sont choisies de façon à ce qu'il existe une possibilité pour qu'elles aboutissent à une solution. La façon d'aboutir à une solution est la décomposition des problèmes, nous rendons donc les symboles à la racine égaux.

Optimisations Une équation n'est pas appliquée consécutivement dans un sens et dans le sens inverse car cela revient à appliquer l'identité.

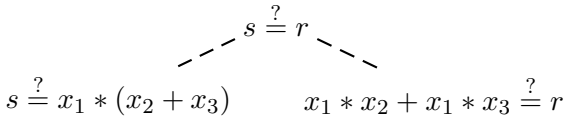
La façon dont sont créées les suites de transformations implique que le même travail est fait plusieurs fois de façon consécutive. En effet, l'application d'une transformation de la forme $e_1.e_2.e_3.e_4$ (où les e_i sont des équations de $\mathcal{E}$) est généralement suivie de l'application d'une transformation $e_1.e_2.e_3.e_5$. Le début étant identique, il est donc souhaitable de factoriser les opérations. Pour cela, les résultats d'une transformation sont stockés. Quand une nouvelle transformation est appliquée, le calcul ne reprend qu'à partir du moment où les deux transformations sont différentes. La reprise des calculs sur les termes qui diffèrent doit donc être réalisée avec les bonnes variables intermédiaires, celles-ci seront elles aussi stockées. Cela permet un gain de temps important.

Nous pouvons ajouter un poids strictement positif à chaque équation ou à chaque orientation de l'équation, pour distinguer les équations coûteuses. Lors de l'application d'une équation, au lieu de décrémenter la quantité d'énergie de un, nous la décrémentons du poids associé à l'équation.

Exemple 3.5.2
Une équation de distribution $x_1 * (x_2 + x_3) = x_1 * x_2 + x_1 * x_3$, coûte plus chère dans le sens



que dans le sens



car cela conduit à une solution où les services réalisent deux multiplications au lieu d'une.

3.5.3 Construction des solutions

Les différents sous-problèmes sont traités de façon indépendante et s'il y a plusieurs équations applicables, plusieurs solutions sont possibles pour résoudre un problème. Nous obtenons donc des solutions sous forme d'arbre et/ou :

- « et » : pour les sous-problèmes issus d'une décomposition ou de l'application d'une équation ;
- « ou » : pour les différentes façons de traiter un problème.

Pour obtenir les solutions sous leur forme finale, il faut distribuer les problèmes et supprimer les variables intermédiaires. Plus la distribution se fait tard, plus la quantité de travail est réduite par factorisation. En effet, en distribuant le problème, le même problème apparaît plusieurs fois. Il est donc traité plusieurs fois. Par contre, si la forme factorisée est conservée, celui-ci n'est traité qu'une unique fois.

Pour supprimer les variables intermédiaires, une première solution consiste à utiliser une matrice pour représenter les liens entre les différentes variables. Sa fermeture est calculée pour obtenir le lien entre les variables du service et les constantes de la requête. Cette solution n'est valide que dans le cas de substitutions avec uniquement des variables et échoue dès que des termes apparaissent.

Pour prendre en compte ce point, nous construisons un graphe qui est ensuite parcouru pour associer les variables aux constantes et aux variables libres et relancer l'algorithme lorsque des termes doivent être confrontés. Comme nous l'avons déjà vu, ces relances sont de deux types. Elles sont effectuées en utilisant, au maximum, la quantité d'énergie restante. Cela nous assure la terminaison du calcul, mais peut nous conduire à refuser des solutions valides. Néanmoins, elles seront obtenues si nous augmentons la quantité d'énergie.

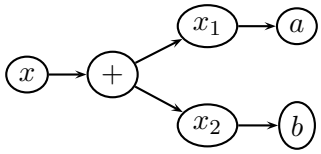
Ces relances se font au moment où une affectation est ajoutée au graphe. Cela permet de ne pas construire un graphe complet quand il y a un échec. Les « conditions » (nouvelles arêtes et nouveaux nœuds) pour que la relance soit correcte sont alors ajoutées au graphe. Quand il y a plusieurs séries de conditions possibles, un nouveau graphe est créé par solution. Le résultat est alors une liste de graphes. En fait, le problème est plus complexe : une liste de triplets est produite, où un triplet est composé : d'un graphe, d'une quantité d'énergie restante et de la liste des affectations qu'il reste à ajouter au graphe.

Dans le graphe, il y a deux sortes de nœuds et d'arêtes : les nœuds « variables » avec les arêtes qui partent de ce nœud signifiant que nous sommes dans la même classe d'équivalence et les nœuds « nœuds » avec les arêtes qui partent de ce nœud signifiant qu'il s'agit du fils du nœud.

Exemple 3.5.3
Si l'analyse d'une forme résolue nous donne la liste d'arêtes suivante :

- $x \rightarrow x_1 + x_2$
- $x_1 \rightarrow a$
- $x_2 \rightarrow b$

le graphe suivant, possédant les deux types de nœuds et d'arêtes, est construit.



L'entrée x est associée à la sortie $a + b$.

Lors de la construction du graphe, si une solution impose une forme particulière à une variable sans qu'elle l'ait réellement, il y a échec : pour le problème $\{(a, x_2), (x_2, x_3 + x_4), (x_3 + x_4, y)\}$, (a, y) n'est pas une solution. En fait, nous n'avons pas de nœud contenant un terme au milieu d'une branche du graphe. En effet, ce n'est valable que si x_2 est une addition, ce qui n'est pas le cas puisque, dans ce cas, x_2 est égal à y. Si y était une addition ($y = y_1 + y_2$), $(x_3 + x_4, y_1 + y_2)$ aurait été décomposé et nous aurions trouvé une structure que nous traitons effectivement correctement.

3.5.4 Options

L'algorithme obtenu est relativement coûteux en terme de temps d'exécution car de nombreuses branches ne conduisent pas à une solution ou construisent des solutions qui ne seront pas intéressantes dans le cas où nous voulons exécuter le résultat (voir exemple 3.5.4).

Nous avons ajouté deux options que l'utilisateur peut choisir d'utiliser ou non. Ces deux options sont :

- Interdire les relances sur la requête ;
- Simplifier la requête.

3.5.4.1 Interdire les relances sur la requête

Cette option permet de ne pas poursuivre vers une solution nécessitant une composition si un des composites est la requête elle-même.

En effet, si nous trouvons une composition de services répondant au problème avec un des composants de cette solution qui est la requête, ce composant, à lui seul, répond au problème. Nous ne perdons donc pas de solutions intéressantes.

Exemple 3.5.4
Si parmi les services initiaux se trouve l'addition ($add(x, y) = x + y$) et la multiplication ($mult(x, y) = x * y$), une façon de résoudre le problème $a * b$ est $mult(a, b)$, mais $add(0, mult(a, b))$ répond également au problème.

Si nous interdisons les relances sur la requête, cette seconde solution n'est pas calculée.

Actuellement, le test de l'égalité entre le terme, sur lequel nous relançons le problème, et de la requête se fait par une comparaison simple, sans utiliser les équations. Nous serons donc peut-être amenés à relancer sur une autre écriture de la requête. Ce choix a été fait car une comparaison modulo les équations est coûteuse.

3.5.4.2 Simplifier la requête

Cette option permet de simplifier la requête quand nous relançons le problème pour une composition.

Certaines règles pourront être orientées et utilisées dans le sens indiqué pour simplifier la requête. Cette simplification est, dans la plupart des cas, intéressante car elle permet de supprimer du travail redondant et de ne pas relancer sur des termes très complexes. Néanmoins, pour certains cas particuliers, les règles utilisées lors de la simplification seront appliquées dans le sens inverse lors de la résolution du problème.

Exemple 3.5.5
Si l'équation $1 * x = x$ *est orientée dans le sens gauche - droite pour la simplification et si nous sommes amenés à relancer sur une requête de la forme* $1 * (A * B)$*, la requête sera simplifiée en* $A * B$ *et le problème relancé avec cette nouvelle requête. Mais si un des services disponibles réalise :* $\alpha * (X * Y)$*, l'équation sera alors appliquée en sens inverse pour trouver une solution.*

De façon globale, sur les exemples pratiques que nous avons exécutés, la combinaison des deux options entraîne un gain de temps et diminue considérablement le nombre de comparaisons, sans pour autant perdre de solutions intéressantes. De plus, dans le cas d'un mécanisme de cache, l'option de simplification permet d'avoir moins de formes possibles pour une requête et donc plus de chance de trouver cette requête dans le cache.

3.6 Un algorithme parallèle et incrémental

3.6.1 Un algorithme parallèle

La structure même de l'algorithme fait de lui un algorithme facilement parallélisable. En effet, tous les sous-problèmes sont traités de façon indépendante. Ils peuvent donc être traités en parallèle. Une barrière de synchronisation est ensuite nécessaire pour mettre les différentes solutions en commun et construire le graphe de suppression des variables intermédiaires.

3.6.2 Un algorithme incrémental

Les résolutions des problèmes sont faites en utilisant exactement n équations. Donc les sous-problèmes sont résolus pour exactement 0 puis 1, puis ... équations. Les solutions des sous-problèmes sont alors recombinées pour que le nombre total d'équations appliquées soit égal à n. Cela a été fait pour pouvoir réutiliser les solutions calculées sur les différents sous-problèmes.

Si nous stockons ces résultats intermédiaires, il nous suffira de faire les calculs pour exactement $n + 1$ équations pour avoir le résultat. Nous n'aurons plus à chercher les solutions qui demandent moins de $n + 1$ équations.

Pour avoir une réelle interactivité avec l'utilisateur et supprimer au maximum l'importance de la quantité d'énergie, un mode « par étapes » est proposé. Il permet de proposer les solutions par niveau d'énergie nécessaire. Pour l'instant, les étapes sont caractérisées par le nombre d'équations appliquées. À chaque étape (où une solution est trouvée), l'utilisateur indique s'il veut poursuivre ou s'il est satisfait des solutions et arrête les recherches.

Il faudrait pouvoir ultérieurement affiner les étapes. Il serait par exemple intéressant de prendre en compte la profondeur de composition ou d'afficher les solutions une à une (actuellement les solutions sont traitées en « bloc »).

Le problème de cette version par étape est le temps perdu (dans le cas où il y a beaucoup de résultats) à faire la comparaison entre les nouveaux résultats obtenus et les

anciens déjà proposés à l'utilisateur. Il faudrait trouver un moyen plus rapide de faire les comparaisons.

De plus, ce mode devra être modifié pour prendre en compte les propriétés de réutilisation des solutions trouvées que nous venons d'évoquer. Actuellement, seules les solutions avec $n + 1$ équations sont calculées, mais les calculs précédents ne sont pas réutilisés. Il faudra en effet trouver un compromis entre le temps de calcul, le temps de recherche parmi les solutions stockées et l'espace nécessaire au stockage.

3.7 Le typage

Dans tout ce qui a été exposé précédemment, le typage n'a pas été évoqué pour alléger la présentation. Or toutes les données que nous manipulons sont typées. Les paramètres d'entrée des procédures sont typés et nous devons respecter ces types. Il a donc fallu modifier cet algorithme pour prendre en compte les types, qui permettent de supprimer des solutions.

3.7.1 Les choix

Plusieurs possibilités s'offrent à nous pour typer les données et nous devons répondre à plusieurs questions : Acceptons nous le polymorphisme ? Quel type de polymorphisme ? Inférence de type ou types explicites et vérification ?

3.7.1.1 Le polymorphisme et l'héritage

Il nous semblait important de posséder la notion de sous-type dans la mesure où cela permet d'exprimer des propriétés différentes sur nos éléments.

Exemple 3.7.1
Par exemple, dans le cas de l'algèbre linéaire, le type $Matrix$ est disponible ainsi que les types $InversibleMatrix$ et $SymmetricMatrix$ qui sont des sous-types du type $Matrix$. Cela nous permet de définir des propriétés sur les éléments que nous manipulons.

La nécessité de l'héritage multiple apparaît également, dans la mesure où plusieurs propriétés peuvent être combinées. Une matrice peut être symétrique et inversible.

Nous avons donc opté pour un typage avec des types ordonnés et un héritage multiple.

Dans [GM92], Goguen, inspiré par [CW85] de Cardelli et Wegner, distingue au moins trois sortes de polymorphisme :

- Polymorphisme ad-hoc ou surcharge.
  Le polymorphisme ad-hoc correspond à la redéfinition d'une même méthode avec une signature différente et indépendante. Par exemple, s'il n'y a pas de lien d'héritage entre Int et $Real$, la redéfinition de $+$ pour les entiers et les réels est de la surcharge.
- Polymorphisme d'inclusion ou sous-typage.
  Toute méthode définie sur un type est définie sur un sous-type.
- Polymorphisme paramétrique ou généricité.

Dans un premier temps, nous avons choisi de ne pas traiter la généricité, mais cela fait partie des améliorations à apporter à l'algorithme. Ce point est discuté plus en détail dans les perspectives. Les deux premiers types de polymorphisme ont été intégrés car ils permettent une grande liberté à l'utilisateur dans l'expression des domaines.

3.7.1.2 Inférence ou vérification ?

Il est ensuite possible de choisir soit de typer fortement un programme et de vérifier que le typage est correct, soit de faire de l'inférence de type pour déduire le type et vérifier s'il en existe effectivement un correct.

La première solution est plus simple mais beaucoup moins flexible et facile d'utilisation pour l'utilisateur qui doit préciser tous les types. De plus, nous acceptons la surcharge des opérateurs et certaines équations peuvent être valides pour tous les opérateurs de même nom. Devoir écrire toutes les versions de l'équation peut devenir très complexe et entraîne des risques d'oublis. Cependant, il peut parfois être important de préciser le type des variables.

Nous avons donc choisi un intermédiaire entre la vérification simple et l'inférence de type. Certaines variables ont un type imposé et d'autres ont un type libre. Dans le cas des types libres, il faut vérifier qu'au moins un type est possible.

Nous devons donc typer des termes dans le cadre de fonctions surchargées et avec sous-typage. Dans [CGL92], Castagna introduit le $\lambda\&$-calcul qui est un λ-calcul typé qui accepte la surcharge. Il propose un système de types qui permet de typer une expression, en lui associant le plus petit type possible (selon l'ordre des types). Pour que ce système soit possible, il y a deux conditions sur la définition des opérateurs :
Si f est défini sur l'ensemble des signatures : $\{U_i \rightarrow V_i\}_{i \in I}$, alors

- $U_i \leq U_j \Rightarrow V_i \leq V_j$
  Nous retrouvons ici les contraintes des signatures hétérogènes avec sous-typage.
- $U_i \Downarrow U_j \Rightarrow \exists z\ (unique)\ \in I,\ U_z = inf\{U_i, U_j\}$,

où $U_i \Downarrow U_j$ dénote l'existence d'un minorant commun pour les types U_i et U_j.

Exemple 3.7.2
Exemples de définition d'opérateurs qui violent les contraintes :

1. *Int < Real*
   *f : Int * Int -> Real*
   *f : Real * Real -> Int*
   *La première condition est violée car Int * Int* $\leq$ *Real * Real mais Real* $\not\leq$ *Int.*
2. *Int < Real*
   *f : Int * Real -> Real*
   *f : Real * Int -> Real*
   *La seconde condition est violée car Inf{Int * Real, Real * Int} = Int * Int et f n'est pas définie pour Int * Int.*

Nous avons choisi d'utiliser le système de types proposé par Castagna et nous vérifions, lors de la définition d'un domaine, que ces contraintes sont bien respectées.

3.7.2 La réalisation

Les variables des services doivent être typées. Cela n'est en rien une contrainte, puisque les informations les plus faciles à obtenir sur les services d'une bibliothèque sont justement les types des paramètres.

Les constantes de la requête doivent également être typées. Cela non plus n'est pas une contrainte importante car l'utilisateur connaît ses données et le problème qu'il veut résoudre.

Lors de la définition des équations, les variables contenues dans celles-ci peuvent ne pas être typées. Nous considérons alors que l'équation est valide pour tous les types possibles, pour toutes les variables, si les deux membres sont bien formés et que les types des deux membres sont compatibles. Les seules variables du problème qui ne sont pas typées sont donc les variables intermédiaires.

Dans la matrice des transformations, nous ne nous limitons plus à un seul opérateur Nop, mais une entrée est fournie pour chaque type. À chaque comparaison, nous vérifions si les types sont compatibles. Dans le cas de types non déterminés, nous ne faisons aucune vérification. C'est seulement à la fin, lors de la construction du graphe de suppression des variables intermédiaires, que nous vérifions qu'il existe bien un type valide pour ces variables.

Cette méthode à l'inconvénient de retarder la détection des erreurs, mais elle ne duplique pas le travail en effectuant les comparaisons pour chaque type possible pour les variables intermédiaires.

3.8 Les limites de cette approche

Le principal défaut de l'algorithme présenté est son manque d'efficacité. Ce problème est lié à plusieurs facteurs :

- D'abord la gestion de la quantité d'énergie fait que nous faisons le travail sur les sous-problèmes (les fils pour la décomposition et les sous-problèmes au niveau des équations) pour un nombre d'équations égal à 0, puis 1 jusqu'à la valeur maximale. À chaque fois, il y a un travail redondant d'effectué.
- Ensuite, le traitement des sous-problèmes de façon totalement indépendante implique que la recherche d'incompatibilités dans les solutions trouvées ne peut s'effectuer qu'une fois que tous les sous-problèmes ont été traités. En propageant ces contraintes au fur et à mesure, les erreurs pourraient être détectées plus rapidement.
- Enfin, le typage des variables intermédiaires effectué uniquement à leur suppression implique que, lors des comparaisons, nous acceptons des types qui pourraient déjà être détectés comme incompatibles. Là aussi, la détection des erreurs est retardée par rapport à ce qu'elle pourrait être.

Ce problème de performance nous conduit à proposer un second algorithme. Celui-ci ne traite plus les problèmes de façon indépendante mais propage des données, comme la quantité d'énergie nécessaire pour résoudre un sous-problème et les contraintes issues de ce sous-problème. Le travail redondant est ainsi diminué et les erreurs sont détectées plus rapidement.

Un autre défaut est l'absence de preuve de complétude. Ceci n'est pas une contrainte forte, mais nous ne pouvons pas être sûrs de ne pas rejeter une solution intéressante (même si

cela ne s'est pas produit dans les exemples considérés). Nous fondons le second algorithme sur la définition constructive des solutions (voir section 2.3, 44) pour avoir un cadre plus adapté à une preuve de complétude.

Chapitre 4

Un algorithme optimisé

Dans la section précédente, nous avons décrit un algorithme qui permet, à partir d'un ensemble de services S, de trouver les solutions qui répondent à la requête r de l'utilisateur. Nous avons aussi vu que la structure même de l'algorithme et la gestion de la quantité d'énergie impliquaient une redondance dans le travail effectué. De plus, la construction du graphe de suppression des variables intermédiaires retarde la détection des erreurs. De même, le typage est uniquement utilisé pour supprimer des solutions et non pour éviter de les calculer.

Dans cette section, nous allons proposer un algorithme plus efficace, qui propage les contraintes au fur et à mesure de leur découverte. Nous allons également prouver qu'il est correct et argumenter au sujet de sa complétude. Contrairement au premier algorithme, celui-ci s'appuie sur la description constructive de l'ensemble des services.

Nous allons, dans un premier temps, présenter un algorithme simple et nous prouverons sa correction et sa complétude pour certaines formes d'équations. Nous étudierons ensuite le cas où les membres des équations sont linéaires par rapport aux variables libres. Nous modifierons l'algorithme pour prendre en compte ces équations. Nous prouverons la correction de cet algorithme et discuterons de sa complétude. Enfin, nous traiterons le cas général. La complexité de l'algorithme sera ensuite calculée. Puis, nous aborderons son implantation et l'intégration du typage. Nous terminerons ce chapitre en parlant de l'interaction entre plusieurs domaines, avant de faire une brève conclusion.

4.1 Résolution du problème de base : une requête et un service

Rappelons que nous souhaitons engendrer tous les services et compositions de services qui répondent à une requête. Pour cela, nous allons, comme dans l'algorithme précédent, effectuer une comparaison élémentaire entre la requête et chaque service. Ces comparaisons élémentaires sont le cœur de l'algorithme. Nous allons les détailler ici.

À partir d'une signature hétérogène avec sous-typage $(S, \leq, \Sigma)$, d'un ensemble d'équations $\mathcal{E}$, d'un service de la forme $\lambda Var(s).s$, avec s un terme sur Σ et d'une requête r, nous allons effectuer une suite de décompositions du problème pour aboutir à des problèmes élémentaires qui vont nous donner le lien entre les variables de s (les paramètres du service) et les constantes du domaine et de la requête ou des variables libres (issues des variables des équations). Ce lien, exprimé sous la forme d'une substitution, va nous indiquer les valeurs des paramètres qui doivent être transmises au service.

4.1.1 Arbre de décomposition

La comparaison de deux termes (s et r) peut être décomposée en plusieurs sous-problèmes : soit en comparant deux à deux les sous-termes de s et r, soit en créant deux sous-problèmes par application d'une équation de $\mathcal{E}$ (nous choisirons une variante de l'équation avec de nouvelles variables).

Comme pour le premier algorithme, un arbre de décomposition est construit par applications successives de décompositions de termes et d'application d'équations jusqu'à obtenir une forme résolue. Toutes les transformations (décomposition et application d'équations) qui peuvent être appliquées sur un problème le sont. En cas de réussite, elles conduiront à différentes solutions. Notre algorithme construit donc toutes les suites de décompositions possibles.

Dans cet algorithme, les différents sous-problèmes ne sont pas traités indépendemment les uns des autres. En effet, les contraintes obtenues sont propagées. Ces contraintes sont des substitutions qui sont appliquées sur les autres sous-problèmes. En effet, traiter les sous-problèmes de façon indépendante puis vérifier que les solutions obtenues sont compatibles, comme dans le premier algorithme, entraîne un retard dans la détection des erreurs (incompatibilité de types, affectation à deux valeurs différentes, ...). Dans cette version, nous prenons en compte les contraintes obtenues sur les sous-problèmes traités précédemment lors du traitement des sous-problèmes suivants.

Décomposition Les fils sont traités dans l'ordre d'apparition dans la liste des fils et les contraintes obtenues sont transférées sur les fils suivants. Notons que les fils peuvent être traités dans un ordre quelconque, si les contraintes obtenues sont reportées sur les fils qui n'ont pas encore été traités. Ce point sera évoqué dans les perspectives.

La représentation graphique de cette décomposition est présentée dans la figure 4.1. Dans cette figure, les A_i sont des sous-arbres et les σ_i les contraintes obtenues en résolvant les sous-problèmes.

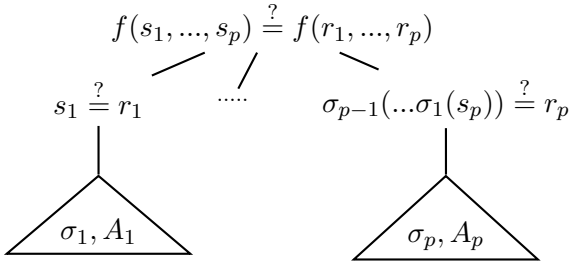


FIG. 4.1 – Décomposition des termes

La substitution (contrainte) associée à cet arbre est : $\sigma_1 \circ ... \circ \sigma_p$.

Application d'une équation Lorsqu'une équation est appliquée, le problème formé de la requête et du second membre est d'abord résolu. Les contraintes sont ensuite transférées sur le premier membre. Le problème formé du service et du premier membre ainsi modifié est alors résolu à son tour.

La représentation graphique de cette application d'équation est présentée dans la figure 4.2. A_1 et A_2 sont des sous-arbres et σ_1 et σ_2 les contraintes obtenues en résolvant les sous-problèmes. Sur cette représentation, le sens de calcul est dans le sens de lecture droite - gauche.

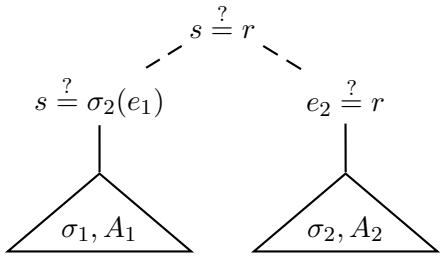


FIG. 4.2 – Application de l'équation (e_1, e_2)

La substitution associée à cet arbre est : σ_1. En effet :

- Les arbres A_i peuvent être vus comme des arbres à part entière, si nous ne considérons que le sous-problème qu'ils résolvent, ou comme des sous-arbres, dans le cas de la résolution du problème global. Cela a un impact sur la substitution associée à cet arbre.
- Pour l'application des équations, dans le cas du traitement du problème global, seule σ_1 est intéressante, car $Dom(\sigma_2) \subset Var(e_2)$ et ces variables n'ont pas de sens dans notre problème initial. Quand le problème est traité de façon globale, la substitution associée à A_2 est $\emptyset$. La substitution σ_2 est celle engendrée par le traitement du sous-problème de façon indépendante.

Des exemples sont donnés dans la suite de ce chapitre (section 4.1.3, page 84).

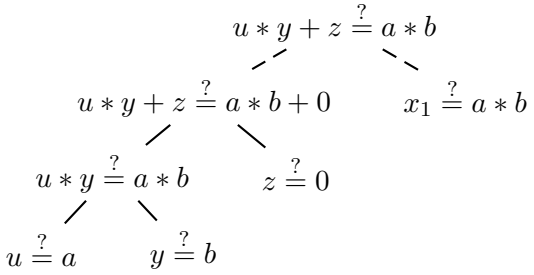
Nous allons maintenant voir quelles sont les substitutions associées aux formes résolues et quelles sont ces formes résolues.

4.1.2 Les formes résolues et la solution associée

Les arbres sous forme résolue sont ceux dont les feuilles $\{s_i \stackrel{?}{=} r_i\}_i$ sont telles que $\forall i$:

- soit s_i et r_i sont deux constantes égales, alors c'est une forme résolue et la substitution associée est $\emptyset$. Si elles sont différentes, il y a échec de la comparaison.
- soit s_i est une variable et la substitution associée est $\{s_i \mapsto r_i\}$.

Exemple 4.1.1
À l'arbre :



est associée la substitution : $\{u \mapsto a,\ y \mapsto b,\ z \mapsto 0\}$

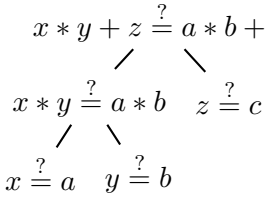
Propriété 4.1.1
Pour tous les problèmes $s \stackrel{?}{=} r$, la substitution σ associée en cas de réussite est telle que $Dom(\sigma) = Var(s) \backslash Var(r)$.

Preuve Les termes sont décomposés jusqu'à aboutir à une forme qui associe à chaque variable une variable, une constante ou un terme. Donc toutes les variables de s seront associées et seront dans $Dom(\sigma)$. Néanmoins, si des variables apparaissent à la fois dans $Var(s)$ et $Var(r)$, nous pouvons engendrer des substitutions de la forme $\{v \mapsto v\}$ et alors v n'apparaîtra pas dans $Dom(\sigma)$. □

4.1.3 Exemples

Les exemples suivants sont les mêmes que ceux qui ont illustré l'algorithme précédent. Cela va permettre de mettre en évidence les différences fondamentales de comportement des deux algorithmes.

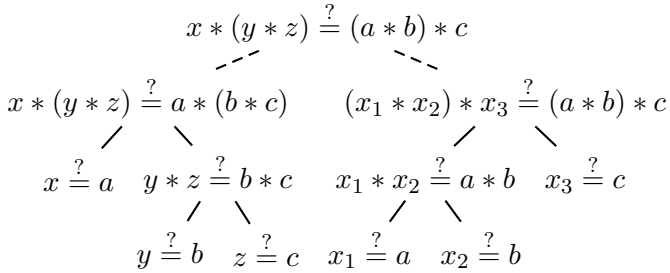
Décomposition Un service fourni est $s(x, y, z) = \lambda x \lambda y \lambda z.x * y + z$, la requête est $r = a * b + c$. Quelles que soient les équations, une solution est obtenue de la façon suivante :



Solution : $r = s(a, b, c)$.

Dans la mesure où chaque variable du service n'apparaît qu'une seule fois, il n'y a pas de différence entre les arbres construits par les deux algorithmes.

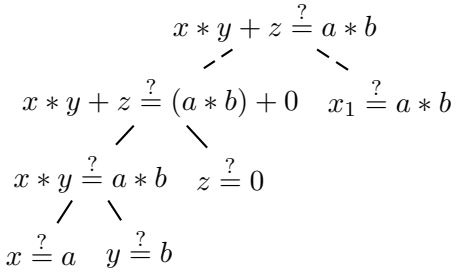
Une équation Un service fourni est $s(x, y, z) = \lambda x \lambda y \lambda z.x * (y * z)$, la requête est $r = (a * b) * c$. Parmi les équations se trouve $(v * u) * w = v * (u * w)$. Une solution est obtenue de la façon suivante :



Solution : $r = s(a, b, c)$.

Nous voyons ici que le graphe de suppression des variables intermédiaires n'est plus nécessaire pour obtenir les solutions. En effet les contraintes sont propagées aux sous-problèmes suivants au fur et à mesure de leur calcul. Nous perdons ainsi l'indépendance du traitement des différents sous-problèmes.

Une autre équation Un service fourni est $s(x, y, z) = \lambda x \lambda y \lambda z.x * y + z$, la requête est $r = a * b$. Parmi les équations se trouve $v = v + 0$. Une solution est obtenue de la façon suivante :

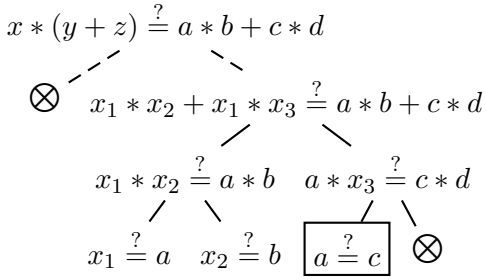


Solution : $r = s(a, b, 0)$.

Dans l'algorithme précédent, nous avions besoin de relancer l'algorithme sur un autre problème. Ceci n'est plus nécessaire dans cet version de l'algorithme, puisque les contraintes sont propagées et que le problème sur lequel devait être relancé l'algorithme ($x * y \stackrel{?}{=} a * b$) apparaît maintenant dans l'arbre.

Composition Il n'y a pas de changement de comportement sur le traitement de la composition.

Échec Un service fourni est $s(x, y, z) = \lambda x \lambda y \lambda z.x * (y + z)$, la requête est $r = a * b + c * d$. Parmi les équations se trouve $i * (j + k) = i * j + i * k$. Une solution est recherchée de la façon suivante :



ÉCHEC

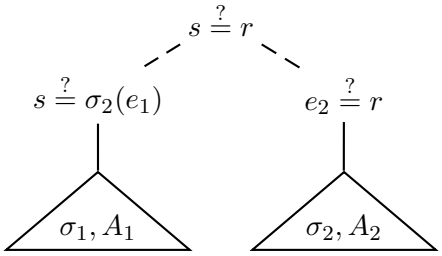
Cet exemple illustre la propagation des contraintes aux frères et la détection des erreurs plus tôt que dans le premier algorithme. Les sous-arbres marqués par $\otimes$ sont ceux que nous ne construisons pas dans cette version de l'algorithme mais que nous construisions dans la version précédente.

4.2 Étude d'un cas simple : $Var(e_1) = Var(e_2)$

Dans un premier temps, nous nous plaçons dans le cas où, pour toutes les équations $(e_1, e_2) \in \mathcal{E}$, $Var(e_1) = Var(e_2)$. Les équations qui sont rejetées sont par exemple :

- $0 * x = 0$
- $x * x^{-1} = I$

Nous partons d'un problème $s \stackrel{?}{=} r$, où $Var(r) = \emptyset$ et cette propriété est conservée sur les nœuds fils. C'est évident pour la décomposition. Pour les équations :



Nous avons $Dom(\sigma_2) = Var(e_2) = Var(e_1)$ et $Im(\sigma_2) = \emptyset$ donc $Var(\sigma_2(e_1)) = \emptyset$.

4.2.1 Preuve de la correction

Pour montrer que l'algorithme engendre un ensemble correct de solutions, il faut montrer que toute solution trouvée par l'algorithme fait partie de l'ensemble des solutions. Il faut donc montrer que, pour s un terme et r une requête, si l'algorithme résout le problème $s \stackrel{?}{=} r$ en créant une substitution σ, il est alors possible de construire une suite de transformations (définies dans la section 2.3, page 44) A telle que :

- $\sigma = \sigma_A$
- $\lambda Var(s).s \xrightarrow{A} r$.

Nous allons montrer que cette propriété est vraie aux feuilles de l'arbre et que si elle est vraie pour tous les sous-problèmes issus d'un nœud, elle est vraie pour le nœud lui-même. Nous montrerons ainsi que la propriété est vraie à la racine.

4.2.1.1 Preuve : aux feuilles

Les feuilles de l'arbre sont sous forme résolue, elles sont donc de la forme :

- $c_1 \stackrel{?}{=} c_2$, où c_1 et c_2 sont deux constantes. La forme est résolue, donc elles sont égales et la substitution associée est $\emptyset$. La propriété est donc vérifiée.
- $v \stackrel{?}{=} c$, où v est une variable et c une constante.
  La substitution associée est $\{v \mapsto c\}$.
  Or :
  - $\lambda v.v \xrightarrow{cste(v,c)} c$
  - $\sigma_{cste(v,c)} = \{v \mapsto c\}$.

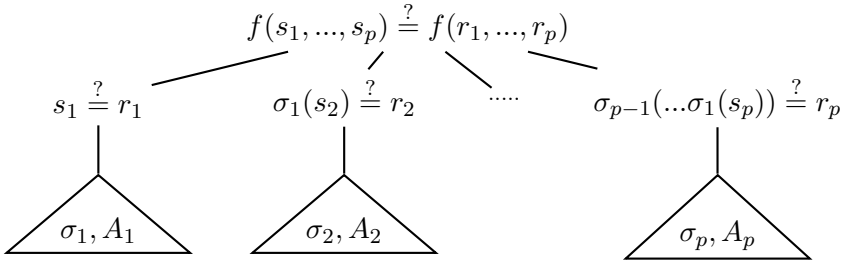
  La propriété est donc vérifiée.
- $v \stackrel{?}{=} t$, où v est une variable et t un terme composé de constantes.
  La substitution associée est $\{v \mapsto t\}$.
  Or :
  - $\lambda v.v \xrightarrow{comp(v,t)} t$
  - $\sigma_{comp(v,t)} = \{v \mapsto t\}$.

  La propriété est donc vérifiée.

La propriété est donc vérifiée aux feuilles de l'arbre.

4.2.1.2 Preuve : aux nœuds

Décomposition



Supposons que la propriété est vraie pour les différents fils, montrons qu'elle est vraie pour $f(s_1, ..., s_p) \stackrel{?}{=} f(r_1, ..., r_p)$. Nous notons $[\![A]\!]$ la suite de transformations associée à l'arbre A.

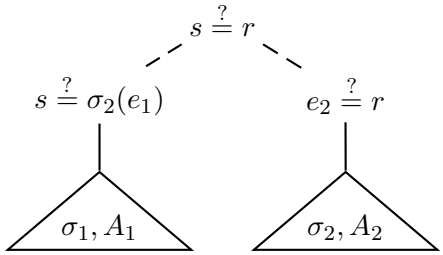
Dans toutes les suites de transformations $[\![A_i]\!]$, les positions d'application des équations devront être préfixées par i dans le cas où nous traiterons le problème global. Ces équations n'agiront donc que sur le fils i et pas sur ses frères. De plus, nous avons vu qu'appliquer une suite de transformations sans transformation eq revient à appliquer la substitution associée. Donc appliquer $[\![A_i]\!]$ reviendra à appliquer $\sigma_{[\![A_i]\!]}$ (et donc σ_i par hypothèse de récurrence) sur les frères du fils i.

	$f(s_1, ..., s_p)$	
$\xrightarrow{[\![A_1]\!]}$	$\lambda.f(r_1, \sigma_1(s_2), ..., \sigma_1(s_p))$	par hypothèse de récurrence sur le premier fils
$\xrightarrow{[\![A_2]\!]}$	$\lambda.f(r_1, r_2, ..., \sigma_2(\sigma_1(s_p)))$	par hypothèse de récurrence sur le deuxième fils
	...	
$\xrightarrow{[\![A_p]\!]}$	$f(r_1, r_2, ..., r_p)$	par hypothèse de récurrence sur le dernier fils

Nous avons donc :
- $\lambda.f(s_1, ..., s_p) \xrightarrow{[\![A_1]\!];...;[\![A_p]\!]} f(r_1, r_2, ..., r_p)$
- $\sigma_{[\![A_1]\!];...;[\![A_p]\!]} = \sigma_{[\![A_1]\!]} \circ ... \circ \sigma_{[\![A_p]\!]} = \sigma_1 \circ ... \circ \sigma_p$.

La propriété est donc vraie.

Application d'une équation



Supposons que la propriété est vraie pour les deux fils, montrons qu'elle est vraie pour $s \stackrel{?}{=} r$.

	$\lambda.s$	
$\xrightarrow{[\![A_1]\!]}$	$\lambda.\sigma_2(e_1)$	par hypothèse de récurrence sur le problème $s \stackrel{?}{=} \sigma_2(e_1)$
$\xrightarrow{eq(\epsilon,\sigma_2,e_1,e_2)}$	$\lambda.\sigma_2(e_2)$	
$\xrightarrow{[\![A_2]\!]_E}$	r	par hypothèse de récurrence sur le problème $e_2 \stackrel{?}{=} r$ et en utilisant la propriété 2.5.1 (page 53)

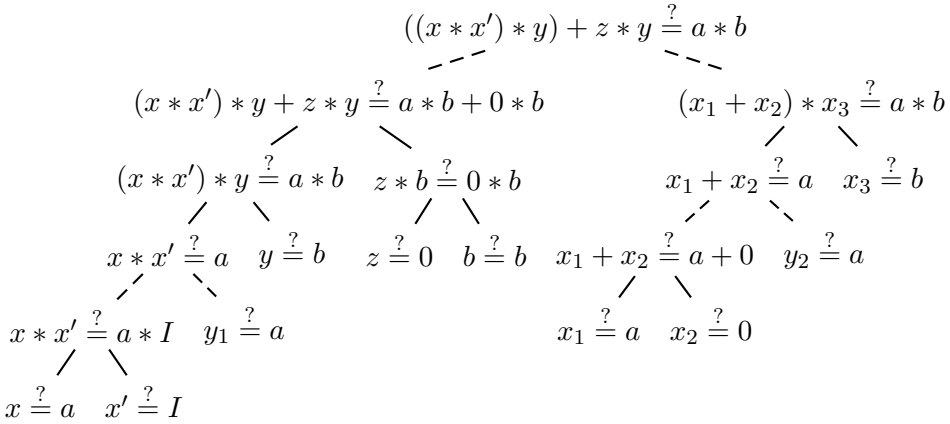
Nous avons donc :

– $\lambda Var(s).s \xrightarrow{[\![A_1]\!];eq(\epsilon,\sigma_2,e_1,e_2);[\![A_2]\!]_E} r$
– $\sigma_{[\![A_1]\!];eq(\epsilon,\sigma_2,e_1,e_2);[\![A_2]\!]_E} = \sigma_{[\![A_1]\!]}$ puisqu'aux transformations eq sont associées des substitutions $\emptyset$. De plus, par hypothèse de récurrence $\sigma_{[\![A_1]\!]} = \sigma_1$.

La propriété est donc vraie.

4.2.1.3 Exemple de suite de transformations associée à un arbre

La transformation engendrée à partir de l'arbre :



est :

1. $cste(x, a)$
2. $cste(x', I)$
3. $eq(11, \{y_1 \mapsto a\}, y_1 * I, y_1)$
4. $cste(y, b)$
5. $cste(z, 0)$
6. $eq(\epsilon, \{x_1 \mapsto a,\ x_2 \mapsto 0,\ x_3 \mapsto b\}, x_1 * x_3 + x_2 * x_3, (x_1 + x_2) * x_3)$
7. $eq(1, \{y_2 \mapsto a\}, y_2 + 0, y_2)$

En l'appliquant sur le service, nous obtenons :

	$\lambda x\ \lambda x' \lambda y\ \lambda z.((x * x') * y) + z * y$
$\xrightarrow{cste(x,a)}$	$\lambda x' \lambda y\ \lambda z.((a * x') * y) + z * y$
$\xrightarrow{cste(x',I)}$	$\lambda y\ \lambda z.((a * I) * y) + z * y$
$\xrightarrow{eq(11,\{y_1 \mapsto a\},y_1 * I,y_1)}$	$\lambda y\ \lambda z.a * y + z * y$
$\xrightarrow{cste(y,b)}$	$\lambda z.a * b + z * b$
$\xrightarrow{cste(z,0)}$	$a * b + 0 * b$
$\xrightarrow{eq(\epsilon,\{x_1 \mapsto a,\ x_2 \mapsto 0,\ x_3 \mapsto b\},x_1 * x_3 + x_2 * x_3,(x_1 + x_2) * x_3)}$	$(a + 0) * b$
$\xrightarrow{eq(1,\{y_2 \mapsto a\},y_2 + 0,y_2)}$	$a * b$

4.2.2 Preuve de la complétude

Pour prouver que l'algorithme est complet, il faut montrer que s'il existe une suite de transformations A, telle que $\lambda Var(s).s \xrightarrow{A} r$, alors l'algorithme construit une substitution

égale à σ_A sur $Var(s)$.

La preuve de la complétude est basée sur un ré-ordonnancement des suites de transformations tout en conservant la même substitution associée. Nous utiliserons une séparation des branches indépendantes par rapport aux équations appliquées. Cette preuve est faite par récurrence sur le nombre d'équations appliquées.

4.2.2.1 Les branches indépendantes

Nous faisons une preuve par récurrence sur le nombre d'équations appliquées. Nous allons donc décomposer notre problème en plusieurs sous-problèmes plus simples, dans lesquels strictement moins de n équations sont appliquées. Pour cela, nous allons isoler une équation (celle à la plus haute position, sur le fils le plus à gauche et apparaissant en premier dans la transformation) et mettre en évidence les différents sous-problèmes.

Nous souhaitons représenter les sous-arbres dont la racine est l'application d'une équation et dont les nœuds parents sont uniquement des décompositions. Nous pouvons construire l'ensemble des positions de ces sous-arbres. Considérons α la plus haute de ces positions et la plus à gauche si plusieurs ont la même hauteur.

Nous souhaitons maintenant décrire l'arbre, privé de α et des nœuds parents de α, comme un ensemble de sous-arbres distincts pour raisonner par récurrence sur ces sous-arbres qui contiennent strictement moins de n équations. Nous appelons ensemble des positions indépendantes les positions de ces sous-arbres car une équation appliquée à la position α n'a pas d'effet sur eux et α étant la position la plus haute où est appliquée une équation, les autres équations appliquées n'agiront que sur un seul de ces sous-arbres.

Nous ajouterons α à cet ensemble. Le sous-arbre à la position α pouvant contenir n équations, il sera par la suite scindé en deux en isolant l'équation évoquée précédemment.

Définition 4.2.1 (Construction de l'ensemble des positions indépendantes)
Construisons l'ensemble de positions indépendantes p_i décrit précédemment.

α est de la forme $i_1 i_2 ... i_n$. Toutes les positions de niveau 1 différentes de i_1 font partie de l'ensemble des p_i. Toutes les positions de niveau 2 de la forme $i_1 i'_2$ et différentes de $i_1 i_2$ font parties de l'ensemble des solutions. Et ainsi de suite jusqu'au niveau n, où toutes les positions de la forme $i_1 i_2 ... i'_n$ (y compris $i_1 i_2 ... i_n$) font partie de l'ensemble des p_i, donc $\alpha \in \{p_1, ..., p_n\}$. Ces positions sont à des niveaux supérieurs ou égaux à α. Par définition de α nous n'avons donc que des décompositions jusqu'à ces points. Les sous-problèmes engendrés en décomposant le problème initial jusqu'à ces positions sont traités avec moins de n équations, puisque n équations sont appliquées au total et celle à la position α ne fait partie d'aucun de ces sous-problèmes. Nous pourrons donc appliquer notre hypothèse de récurrence sur ces sous-problèmes.

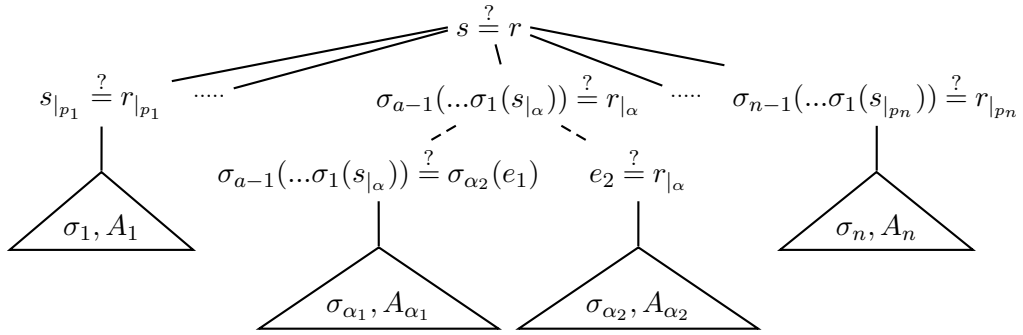
α étant dans l'ensemble des positions indépendantes, $\exists a$ tel que $\alpha = p_a$.

Exemple 4.2.1
Si α = 222, l'ensemble des positions indépendantes est :

$$\{1, 21, 221, 222, ..., 22n_1, 23, ..., 2n_2, 3, ..., n_3\}.$$

Dans le cas où $\alpha = \epsilon$, seul α appartient à l'ensemble des positions indépendantes.

Par abus de notation, la structure de l'arbre que l'algorithme construit est représentée de la façon suivante :

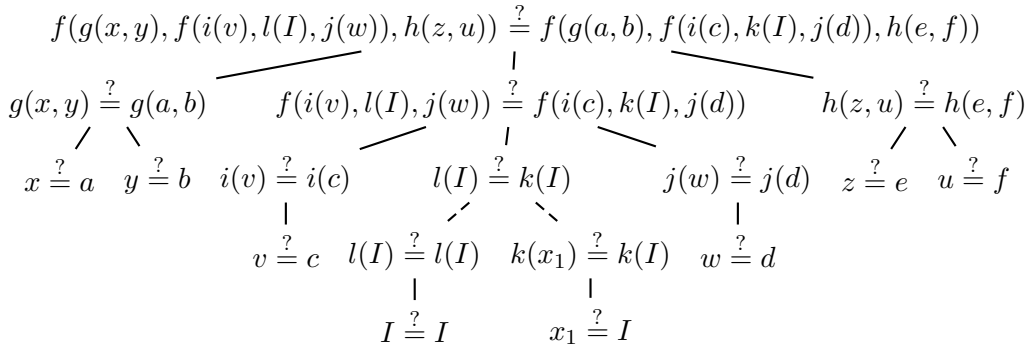


Tous les liens qui partent du problème initial ne sont constitués que de décompositions et sont de taille variable, inférieure ou égale à celle de α.

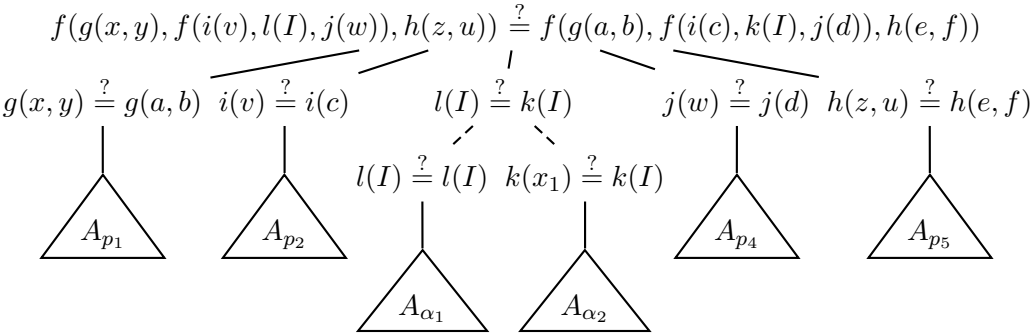
Exemple 4.2.2

Soit le domaine défini sur un type unique et dont les opérateurs sont : f (arité 3), g (arité 2), i (arité 1), j (arité 1), l (arité 1), k (arité 1), h (arité 2) et I une constante. L'ensemble des équations est $\{l(x_1) = k(x_1)\}$.

Soit l'arbre :



Ici, la position la plus haute où une équation est appliquée est $\alpha = 22$*, l'ensemble des positions indépendantes est* $\{1, 21, 22, 23, 3\}$*. Par abus de notation, l'arbre est représenté de la façon suivante :*



où $A_{p_1} = A_1$ *:* $g(x,y) \stackrel{?}{=} g(a,b)$, $x \stackrel{?}{=} a$, $y \stackrel{?}{=} b$

$A_{p_2} = A_{21}$ *:* $i(v) \stackrel{?}{=} i(c)$, $v \stackrel{?}{=} c$

$A_{\alpha_1} = A_{22_1}$ *:* $l(I) \stackrel{?}{=} l(I)$, $I \stackrel{?}{=} I$

$A_{\alpha_2} = A_{22_2}$ *:* $k(x_1) \stackrel{?}{=} k(I)$, $x_1 \stackrel{?}{=} I$

$A_{p_4} = A_{23}$: $j(w) \stackrel{?}{=} j(d)$; $w \stackrel{?}{=} d$

$A_{p_5} = A_2$: $h(z,u) \stackrel{?}{=} h(e,f)$; $z \stackrel{?}{=} e$; $u \stackrel{?}{=} f$

Toujours par abus de notation, le service s est représenté de la façon suivante ($s_{|_{p_1}}$ ... $s_{|_\alpha}$ ... $s_{|_{pn}}$).

Exemple 4.2.3
Pour l'exemple précédent, $f(g(x,y), f(i(v), l(I), j(w)), h(z,u))$ *est noté* $(g(x,y),\ i(v),\ l(I),\ j(w),\ h(z,u))$.

Nous faisons de même pour la requête.

Ces notations « abusives » seront celles utilisées dans la preuve de complétude pour représenter les termes et les arbres.

4.2.2.2 Preuve : l'hypothèse de récurrence

Soient s un terme sur Σ et r une requête, s'il existe une suite de transformations A telle que $\lambda Var(s).s \xrightarrow{A} r$ et A possède n transformations eq, alors l'algorithme permet de construire un arbre sous forme résolue et la substitution engendrée est égale à σ_A sur $Var(s)$.

4.2.2.3 Preuve : sans équation

Montrons que la propriété est vraie dans le cas où il n'y a aucune équation appliquée.
Preuve par récurrence sur la profondeur du service.

Le service s est une variable

- Si r est une constante, soit A une transformation sans équation telle que $\lambda s.s \xrightarrow{A} r$. Nous avons alors $\sigma_{A|Var(s)} = \{s \mapsto r\}$, puisque nous avons montré que dans le cas sans équation $\sigma_A(s) = r$.
  Le problème $s \stackrel{?}{=} r$ est sous forme résolue et la substitution associée est $\{s \mapsto r\}$. La propriété est donc vérifiée.
- Si r est un terme, soit A une transformation sans équation telle que $\lambda s.s \xrightarrow{A} r$, nous avons alors $\sigma_{A|Var(s)} = \{s \mapsto r\}$.
  Le problème $s \stackrel{?}{=} r$ est sous forme résolue et la substitution associée est $\{s \mapsto r\}$. La propriété est donc vérifiée.

Le service est un terme Supposons que la propriété est vraie pour des termes de profondeur inférieure à m, montrons qu'elle l'est pour des termes de profondeur m.

s est de profondeur ≥ 1, donc s est de la forme $f(s_1, ..., s_p)$. Comme il existe une solution au problème et qu'aucune équation n'est appliquée, r est aussi de la forme $f(r_1, ..., r_p)$.

Soit A une suite de transformations (sans transformation eq) telle que :

$$\lambda Var(s).s \xrightarrow{A} r$$

Montrons que l'algorithme trouve une solution égale à σ_A sur $Var(s)$.

Nous avons $\lambda Var(s).s \xrightarrow{A} r$.

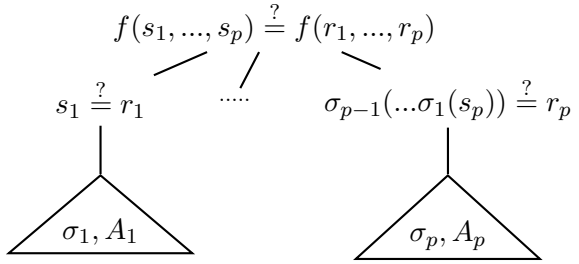
1. Si $\sigma_1 = \sigma_{A|Var(s_1)}$, nous avons alors :
   $\lambda Var(s_1).s_1 \xrightarrow{A_{\sigma_1}} r_1$ et
   $\lambda Var(s).f(s_1, s_2, ..., s_p) \xrightarrow{A_{\sigma_1}} f(r_1, \sigma_1(s_2), ..., \sigma_1(s_p))$.

2. Si $\sigma_1 = \sigma_{A|Var(\sigma_1(s_2))}$, nous avons alors :
   $\lambda Var(\sigma_1(s_2)).\sigma_1(s_2) \xrightarrow{A_{\sigma_2}} r_2$ et
   $f(r_1, \sigma_1(s_2), ..., \sigma_1(s_p)) \xrightarrow{A_{\sigma_2}} f(r_1, r_2, ..., \sigma_2(\sigma_1(s_p)))$
   Notons que $Var(\sigma_1(s_2)) = Var(e_2) \backslash Var(e_1)$ car $Im(\sigma_1) = \emptyset$ et $Dom(\sigma_1) = Var(s_1)$.

3. ...

   Nous allons ainsi construire σ_1,...,σ_p telles que :
   - $\lambda Var(\sigma_{i-1}(...(\sigma_1(s_i)))).\sigma_{i-1}(...(\sigma_1(s_i))) \xrightarrow{A_{\sigma_i}} r_i$
   - $\lambda Var(s) \xrightarrow{A_{\sigma_1};...;A_{\sigma_p}} r$
   - $\sigma_{A|Var(s)} = \sigma_1 \cup ... \cup \sigma_p = \sigma_1 \circ ... \circ \sigma_p$ car les substitutions ont des domaines disjoints et que leurs images sont vides.

Par hypothèse de récurrence, nous pouvons construire un arbre valide pour les problèmes :
$\sigma_{i-1}(...(\sigma_1(s_i))) \stackrel{?}{=} r_i$
et les substitutions associées sont σ_1, ..., σ_p. Nous pouvons donc construire un arbre de la forme :



qui est une forme valide de notre algorithme. La substitution associée est $\sigma_1 \circ ... \circ \sigma_p$ qui est égale à $\sigma_{A|Var(s)}$. Comme l'algorithme construit tous les arbres de cette forme, il trouve donc une substitution égale à σ_A sur $Var(s)$. La propriété est donc vraie.

4.2.2.4 Preuve : avec équations

Supposons que la propriété est vraie pour strictement moins de n équations appliquées, montrons qu'elle est vraie pour n équations appliquées.

Supposons qu'il existe une suite A de transformations telle que $\lambda Var(s).s \xrightarrow{A} \lambda Var(r).r$ et qui engendre la substitution σ_A. Montrons que notre algorithme construit un arbre qui engendre une substitution égale à σ_A sur $Var(s)$.

Soit α la plus haute position où est appliquée une équation. Construisons l'ensemble des positions indépendantes $\{p_1, ..., p_m\}$ comme défini précédemment. $\alpha \in \{p_1, ...p_m\}$, soit a tel que $\alpha = p_a$.

Dans l'ensemble A_E, les équations de ces différentes branches sont séparées. Notons A_{E_i} les différentes suite de transformations sur ces branches. Dans chaque A_{E_i}, l'ordre d'application des équations de A_E est conservé.

Comme pour le cas sans équation, séparons $\sigma_{A|Var(s)}$ en $\sigma_1 \cup ... \cup \sigma_m = \sigma_1 \circ ... \circ \sigma_m$, où

- $Dom(\sigma_1) = Var(s_{|_{p_1}})$
- $Dom(\sigma_2) = Var(s_{|_{p_2}}) \backslash Var(s_{|_{p_1}})$
- ...

1. Nous savons que :
   $\lambda.\sigma_A(s_{|_{p_1}}) \xrightarrow{A_{E_1}} r_{|_{p_1}}$, d'après la propriété de ré-ordonnancement (propriété 2.5.1, page 53) et puisque seules des équations de A_{E_1} agissent sur $\sigma_A(s_{|_{p_1}})$
   donc $\lambda.s_{|_{p_1}} \xrightarrow{A_{\sigma_1}} \lambda.\sigma_1(s_{|_{p_1}}) = \lambda.\sigma_A(s_{|_{p_1}}) \xrightarrow{A_{E_1}} r_{|_{p_1}}$

   Donc
   $$\begin{array}{ll}
   & \lambda.\ (s_{|_{p_1}}, s_{|_{p_2}}, ..., s_{|_{p_{a-1}}}, s_{|_\alpha}, s_{|_{p_{a+1}}}, ..., s_{|_{p_m}}) \\
   \xrightarrow{A_{\sigma_1}} & \lambda.(\sigma_1(s_{|_{p_1}}), \sigma_1(s_{|_{p_2}}), ..., \sigma_1(s_{|_{p_{a-1}}}), \sigma_1(s_{|_\alpha}), \sigma_1(s_{|_{p_{a+1}}}), ..., \sigma_1(s_{|_{p_m}})) \\
   \xrightarrow{A_{E_1}} & \lambda.\ (r_{|_{p_1}}, \sigma_1(s_{|_{p_2}}), ..., \sigma_1(s_{|_{p_{a-1}}}), \sigma_1(s_{|_\alpha}), \sigma_1(s_{|_{p_{a+1}}}), ..., \sigma_1(s_{|_{p_m}})) \\
   \xrightarrow{A_{\sigma_2 \cup ... \cup \sigma_m}} & \lambda.\ (r_{|_{p_1}}, \sigma_A(s_{|_{p_2}}), ..., \sigma_A(s_{|_{p_{a-1}}}), \sigma_A(s_{|_\alpha}), \sigma_A(s_{|_{p_{a+1}}}), ..., \sigma_A(s_{|_{p_m}})) \\
   \xrightarrow{A_{E_2}; ...; A_{E_m}} & (r_{|_{p_1}}, r_{|_{p_2}}, ..., r_{|_{p_{a-1}}}, r_{|_\alpha}, r_{|_{p_{a+1}}}, ..., r_{|_{p_m}})
   \end{array}$$
   En effet, pour tout $i \neq 1$:
   $\lambda.\sigma_A(s_{|_{p_i}}) \xrightarrow{A_E} \lambda.r_{|_{p_i}}$
   Or E_1 n'agit pas sur ces fils, donc :
   $\lambda.\sigma_A(s_{|_{p_i}}) \xrightarrow{A_{E_2}; ...; A_{E_m}} \lambda.r_{|_{p_i}}$
2. Nous savons que :
   $Var(\sigma_1(s_{|_{p_2}})) = Var(s_{|_{p_2}}) \backslash Var(s_{|_{p_1}}) = Dom(\sigma_2)$
   donc $\sigma_2(\sigma_1(s_{|_{p_2}})) = \sigma_A(s_{|_{p_2}})$
   Or $\lambda.\sigma_A(s_{|_{p_2}}) \xrightarrow{A_{E_2}} r_{|_{p_2}}$
   donc $\lambda.\sigma_1(s_{|_{p_2}}) \xrightarrow{A_{\sigma_2}} \lambda.\sigma_2(\sigma_1(s_{|_{p_2}})) = \lambda.\sigma_A(s_{|_{p_2}}) \xrightarrow{A_{E_2}} r_{|_{p_2}}$

   Donc
   $$\begin{array}{ll}
   & \lambda.\ (s_{|_{p_1}}, s_{|_{p_2}}, ..., s_{|_{p_{a-1}}}, s_{|_\alpha}, s_{|_{p_{a+1}}}, ..., s_{|_{p_m}}) \\
   \xrightarrow{A_{\sigma_1}} & \lambda.(\sigma_1(s_{|_{p_1}}), \sigma_1(s_{|_{p_2}}), ..., \sigma_1(s_{|_{p_{a-1}}}), \sigma_1(s_{|_\alpha}), \sigma_1(s_{|_{p_{a+1}}}), ..., \sigma_1(s_{|_{p_m}})) \\
   \xrightarrow{A_{E_1}} & \lambda.\ (r_{|_{p_1}}, \sigma_1(s_{|_{p_2}}), ..., \sigma_1(s_{|_{p_{a-1}}}), \sigma_1(s_{|_\alpha}), \sigma_1(s_{|_{p_{a+1}}}), ..., \sigma_1(s_{|_{p_m}})) \\
   \xrightarrow{A_{\sigma_2}} & \lambda.\ (r_{|_{p_1}}, \sigma_2(\sigma_1(s_{|_{p_2}})), ..., \sigma_2(\sigma_1(s_{|_{p_{a-1}}})), \sigma_2(\sigma_1(s_{|_\alpha})), \sigma_2(\sigma_1(s_{|_{p_{a+1}}})), ..., \sigma_2(\sigma_1(s_{|_{p_m}}))) \\
   \xrightarrow{A_{E_2}} & \lambda.\ (r_{|_{p_1}}, r_{|_{p_2}}, ..., \sigma_2(\sigma_1(s_{|_{p_{a-1}}})), \sigma_2(\sigma_1(s_{|_\alpha})), \sigma_2(\sigma_1(s_{|_{p_{a+1}}})), ..., \sigma_2(\sigma_1(s_{|_{p_m}}))) \\
   \xrightarrow{A_{\sigma_3 \cup ... \cup \sigma_m}} & \lambda.\ (r_{|_{p_1}}, r_{|_{p_2}}, ..., \sigma_A(s_{|_{p_{a-1}}}), \sigma_A(s_{|_\alpha}), \sigma_A(s_{|_{p_{a+1}}}), ..., \sigma_A(s_{|_{p_m}})) \\
   \xrightarrow{A_{E_3}; ...; A_{E_m}} & (r_{|_{p_1}}, r_{|_{p_2}}, ..., r_{|_{p_{a-1}}}, r_{|_\alpha}, r_{|_{p_{a+1}}}, ..., r_{|_{p_m}})
   \end{array}$$

3. ...

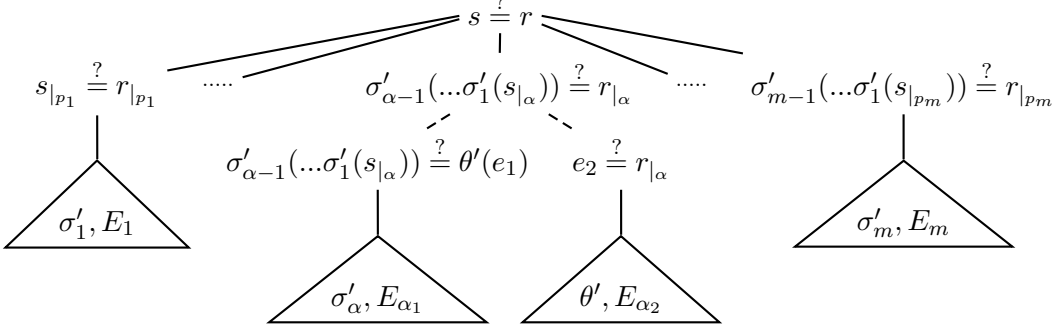
   Notons qu'il s'agit bien évidemment d'une récurrence développée mais l'écriture formelle de la récurrence serait beaucoup plus lourde.

   Découpons A_{E_a} en $A_{E_{\alpha_1}}; eq(\alpha, \theta, e_1, e_2); A_{E_{\alpha_2}}$.
   Nous avons alors la suite de transformations représentée sur la figure 4.1 (page 95).
   – Nous avons vu que pour tout i :
   $\lambda.\sigma_{i-1}(...(\sigma_1(s_{|_{p_i}}))) \xrightarrow{A_{\sigma_i}} \lambda.\sigma_i(...(\sigma_1(s_{|_{p_i}}))) = \lambda.\sigma_A(s_{|_{p_i}}) \xrightarrow{A_{E_i}} r_{|_{p_i}}$.
   Par hypothèse de récurrence, pour tout $i \neq \alpha$, nous savons traiter le problème $\sigma_{i-1}(...(\sigma_1(s_{|_{p_i}}))) \stackrel{?}{=} r_{|_{p_i}}$ et nous construisons une substitution σ'_i égale à σ_i sur $Var(\sigma_{i-1}(...(\sigma_1(s_{|_{p_i}}))))$.

   – Nous avons $\lambda.\sigma_\alpha(...\sigma_1(s_{|_{p_{a+1}}})) \xrightarrow{A_{\sigma_\alpha}; A_{E_{\alpha_1}}} \lambda.\theta(e_1)$.
   Par hypothèse de récurrence
   – nous savons donc résoudre le problème $\sigma_\alpha(...\sigma_1(s_{|_{p_{a+1}}})) \stackrel{?}{=} \theta(e_1)$
   – la substitution calculée σ'_α est égale à σ_α sur $Var(\sigma_\alpha(...\sigma_1(s_{|_{p_{a+1}}})))$.

   – Nous avons $\lambda Var(\theta(e_2)).\theta(e_2) \xrightarrow{A_{E_{\alpha_2}}} r_{|_\alpha}$, donc :
   $\lambda Var(e_2).e_2 \xrightarrow{A_{\theta(e_2)|Var(e_2)}} \lambda Var(\theta(e_2)).\theta(e_2) \xrightarrow{A_{E_{\alpha_2}}} r_{|_\alpha}$.
   Par hypothèse de récurrence, nous savons donc traiter $e_2 \stackrel{?}{=} r_{|_\alpha}$ et la substitution calculée θ' est égale à θ sur $Var(e_1)$.

   Nous pouvons donc construire un arbre de la forme :



Cet arbre est une forme valide pour notre algorithme. Comme l'algorithme construit toutes les formes valides, nous allons donc trouver la solution $\sigma'_1 \circ ... \circ \sigma'_m = \sigma'_1 \cup ... \cup \sigma'_m = \sigma_{A|Var(s)}$.

Notre algorithme est donc complet dans le cas où, pour toutes les équations $(e_1, e_2) \in \mathcal{E}$, $Var(e_1) = Var(e_2)$.

4.3 Extension à $Var(e_1) \subset Var(e_2)$

Les propriétés précédentes peuvent être étendues au cas d'équations $e_1 = e_2$ appliquées dans un sens tel que $Var(e_1) \subseteq Var(e_2)$.

Correction La propriété $Var(e_1) = Var(e_2)$ intervient au niveau de la preuve de la correction au moment de l'application d'une équation, pour s'assurer que $\theta(e_1)$ ne contient

$$
\begin{array}{lllllllllll}
 & \lambda. & (s_{|_{p_1}} & , & s_{|_{p_2}} & ,..., & s_{|_{p_{a-1}}} & , & s_{|_\alpha} & , & s_{|_{p_{a+1}}} & ,..., & s_{|_{p_m}}) \\
\xrightarrow{A_{\sigma_1}} & \lambda. & (\sigma_1(s_{|_{p_1}}) & , & \sigma_1(s_{|_{p_2}}) & ,..., & \sigma_1(s_{|_{p_{a-1}}}) & , & \sigma_1(s_{|_\alpha}) & , & \sigma_1(s_{|_{p_{a+1}}}) & ,..., & \sigma_1(s_{|_{p_m}})) \\
\xrightarrow{A_{E_1}} & \lambda. & (r_{|_{p_1}} & , & \sigma_1(s_{|_{p_2}}) & ,..., & \sigma_1(s_{|_{p_{a-1}}}) & , & \sigma_1(s_{|_\alpha}) & , & \sigma_1(s_{|_{p_{a+1}}}) & ,..., & \sigma_1(s_{|_{p_m}})) \\
\xrightarrow{A_{\sigma_2}} & \lambda. & (r_{|_{p_1}} & , \sigma_2(\sigma_1(s_{|_{p_2}})) & ,..., & \sigma_2(\sigma_1(s_{|_{p_{a-1}}})) & , & \sigma_2(\sigma_1(s_{|_\alpha})) & , & \sigma_2(\sigma_1(s_{|_{p_{a+1}}})) & ,..., & \sigma_2(\sigma_1(s_{|_{p_m}}))) \\
\xrightarrow{A_{E_2}} & \lambda. & (r_{|_{p_1}} & , & r_{|_{p_2}} & ,..., & \sigma_2(\sigma_1(s_{|_{p_{a-1}}})) & , & \sigma_2(\sigma_1(s_{|_\alpha})) & , & \sigma_2(\sigma_1(s_{|_{p_{a+1}}})) & ,..., & \sigma_2(\sigma_1(s_{|_{p_m}}))) \\
 & ... & & & & & & & & & & & \\
\xrightarrow{A_{\sigma_\alpha}} & \lambda. & (r_{|_{p_1}} & , & r_{|_{p_2}} & ,..., & r_{|_{p_{a-1}}} & , \sigma_\alpha(...\sigma_1(s_{|_\alpha})) & , \sigma_\alpha(...\sigma_1(s_{|_{p_{a+1}}})) & ,..., & \sigma_\alpha(...\sigma_1(s_{|_{p_m}}))) \\
\xrightarrow{A_{E_{\alpha_1}}} & \lambda. & (r_{|_{p_1}} & , & r_{|_{p_2}} & ,..., & r_{|_{p_{a-1}}} & , \theta(e_1) & , \sigma_\alpha(...\sigma_1(s_{|_{p_{a+1}}})) & ,..., & \sigma_\alpha(...\sigma_1(s_{|_{p_m}}))) \\
\xrightarrow{eq(\alpha,\theta,e_1,e_2)} & \lambda. & (r_{|_{p_1}} & , & r_{|_{p_2}} & ,..., & r_{|_{p_{a-1}}} & , \theta(e_2) & , \sigma_\alpha(...\sigma_1(s_{|_{p_{a+1}}})) & ,..., & \sigma_\alpha(...\sigma_1(s_{|_{p_m}}))) \\
\xrightarrow{A_{E_{\alpha_2}}} & \lambda. & (r_{|_{p_1}} & , & r_{|_{p_2}} & ,..., & r_{|_{p_{a-1}}} & , r_{|_\alpha} & , \sigma_\alpha(...\sigma_1(s_{|_{p_{a+1}}})) & ,..., & \sigma_\alpha(...\sigma_1(s_{|_{p_m}}))) \\
 & ... & & & & & & & & & & & \\
\xrightarrow{A_{\sigma_m}} & \lambda. & (r_{|_{p_1}} & , & r_{|_{p_2}} & ,..., & r_{|_{p_{a-1}}} & , r_{|_\alpha} & , r_{|_{p_{a+1}}} & ,..., & \sigma_m(...\sigma_1(s_{|_{p_m}}))) \\
\xrightarrow{A_{E_m}} & & (r_{|_{p_1}} & , & r_{|_{p_2}} & ,..., & r_{|_{p_{a-1}}} & , r_{|_\alpha} & , r_{|_{p_{a+1}}} & ,..., & r_{|_{p_m}}) \\
\end{array}
$$

TAB. 4.1 – Réordonnancement de la transformation

aucune variable. Cela est le cas car $Dom(\theta) = Var(e_2)$. Si $Var(e_1) \subset Var(e_2)$, cette propriété est donc vérifiée.

Donc dans le cas où nous appliquons des équations de telle sorte que $Var(e_1) \subset Var(e_2)$, les solutions trouvées seront correctes.

Complétude La propriété $Var(e_1) = Var(e_2)$ intervient au niveau de la preuve de complétude pour s'assurer que le θ calculé en comparant e_2 et $r_{|\alpha}$ et bien le même sur $Var(e_1)$, que celui qui apparaît dans la suite de transformation.

Une fois encore l'inclusion $Var(e_1) \subseteq Var(e_2)$ est suffisante.

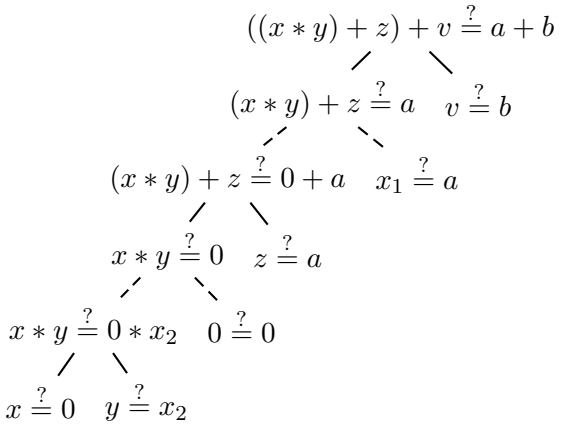
4.4 Étude d'un cas intermédiaire

Nous venons de voir que dans les cas où toutes les équations possèdent les mêmes variables dans les deux membres ou dans le cas où les équations sont appliquées dans un sens précis et que les variables d'un membre sont incluses dans celles de l'autre membre, l'algorithme est correct et complet. Néanmoins, cela restreint beaucoup l'ensemble des équations et nous souhaitons ne pas imposer de contraintes sur ces équations.

Voyons ce qui se passe quand nous avons des ensembles de variables différents. Le problème intervient au niveau de l'application des équations, car $\sigma_2(e_1)$ contient encore des variables : celles de $Var(e_1) \backslash Var(e_2)$. Nous avons alors un problème d'unification équationnelle à résoudre, avec des variables différentes dans les deux membres.

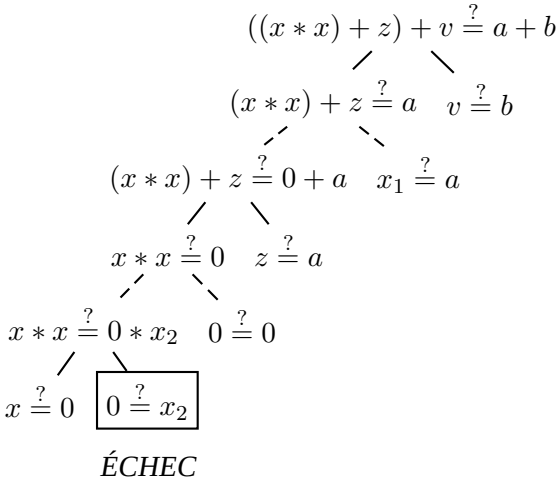
Nous pouvons décider de traiter ces variables libres comme des constantes, auquel cas l'algorithme reste correct, mais nous perdons la complétude, puisque nous perdons les solutions où ces variables doivent prendre des valeurs particulières.

Exemple 4.4.1
Solution que nous trouverions :



La substitution associée est : $\{x \mapsto 0, y \mapsto x_2, z \mapsto a, v \mapsto b\}$.

Exemple 4.4.2
Solution que nous rejetterions :

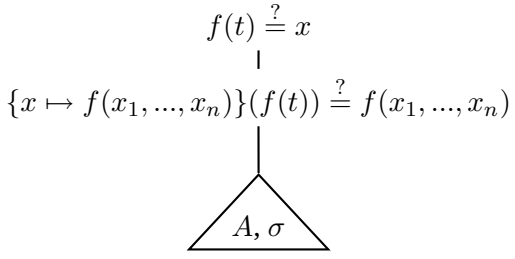


x_2 étant considérée comme une constante et les constantes étant différentes deux à deux, nous rejetons $0 \stackrel{?}{=} x_2$.

Ceci n'est bien sûr pas acceptable, nous avons donc décidés de traiter ces variables comme des variables à part entière et donc de leur associer des valeurs. Cela implique que nous devons propager ces valeurs pour détecter les erreurs. Cela n'est nécessaire que si la variable apparaît deux fois dans le terme.

Traitons d'abord le cas où e_1 est linéaire pour les variables de $Var(e_1) \backslash Var(e_2)$. Nous refusons donc des équations de la forme $x * x^{-1} = I$ (par contre si elle est appliquée dans le sens inverse, elle sera acceptée car $Var(I) \subset Var(x * x^{-1})$, voir 4.2.2.4, 94), mais acceptons celles de la forme $0 * x = 0$.

Nous venons de voir que les contraintes sur ces variables n'ont pas besoin d'être propagées, donc la décomposition des termes et l'application des équations ne sont pas modifiées. Par contre, ces variables libres pouvant prendre une forme quelconque nous devons ajouter une règle de transformation :



La substitution solution sera $\{x \mapsto f(x_1, ..., x_n)\} \circ \sigma$

Cette transformation ne peut s'appliquer que dans le cas où x est une variable libre et les x_i doivent être des nouvelles variables, elles-mêmes considérées comme des variables libres.

Les formes résolues et les substitutions associées changent. Les formes résolues sont maintenant celles de la forme $s \stackrel{?}{=} r$, où :

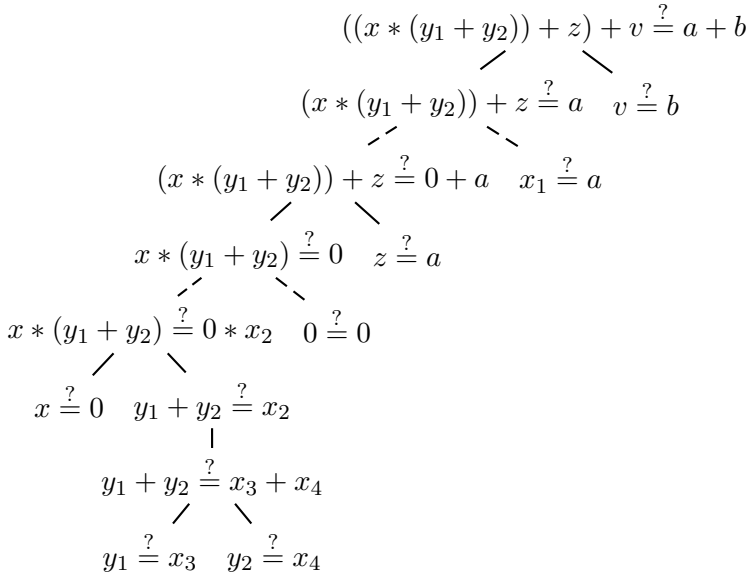
- s est une constante et r est une constante. Si les deux constantes sont différentes, il y a échec. La substitution associée est $\emptyset$.
- s est une constante et r est une variable. La substitution associée est $\{r \mapsto s\}$. Nous sommes ici dans le cas d'une variable libre qui doit être contrainte.

– s est une variable. La substitution associée est $\{s \mapsto r\}$.

Nous conservons la propriété $Dom(\sigma_{|Var(s)}) = Var(s)$. Les variables de s et r étant différentes, nous sommes sûrs que les assignations de deux variables apparaissent effectivement dans la substitution. Par contre, dans σ, nous voyons aussi apparaître les variables libres.

Exemple 4.4.3
L'exemple 4.4.2 est alors sous une forme résolue et la substitution associée est :
$\{x \mapsto 0, x_2 \mapsto 0, z \mapsto a, v \mapsto b\}$.

Exemple 4.4.4
Exemple illustrant l'utilité de la nouvelle transformation :



La substitution associée est : $\{x \mapsto 0, x_2 \mapsto x_3 + x_4, y_1 \mapsto x_3, y_2 \mapsto x_4, z \mapsto a, v \mapsto b\}$.

Dans l'exemple, la variable x_2 est une variable libre, elle peut donc prendre une forme quelconque. Par application de la nouvelle transformation, elle sera en fait affectée à l'addition de deux autres variables libres ($x_3 + x_4$).

La substitution solution, indique que si x est affecté à 0, z à a, v à b et quelque soit les valeurs de y_1 et y_2 le service répondra à la requête.

4.4.1 Preuve de la correction

Nous reprenons le même principe que pour prouver la correction de l'algorithme précédent, mais cette fois la propriété que nous voulons montrer est :
Pour s un terme et r une requête, si l'algorithme résout le problème $s \stackrel{?}{=} r$ en créant une substitution σ, il est alors possible de construire une suite de transformations A telle que :

– $\sigma_{|Var(s)} = \sigma_{A|Var(s)}$

– $\lambda Var(s).s \xrightarrow{A} \lambda Var(\theta(r)).\theta(r)$ avec $Dom(\theta) \subset Var(r)$ et $Im(\theta) = \emptyset$.

Comme à la racine $Var(r) = \emptyset$, si la propriété est vraie à la racine nous aurons bien la correction de notre algorithme.

4.4.1.1 Preuve : aux feuilles

Les feuilles de l'arbre sont sous formes résolues, elles sont donc de la forme :

- $c_1 \stackrel{?}{=} c_2$, où c_1 et c_2 sont deux constantes égales car nous avons une forme résolue et la substitution associée est alors $\emptyset$. La propriété est donc vérifiée avec $\theta = \emptyset$.
- $v \stackrel{?}{=} c$, où v est une variable et c une constante.
  La substitution associée est $\{v \mapsto c\}$.
  Or :
  - $\lambda v.v \xrightarrow{cste(v,c)} c$
  - $\sigma_{cste(v,c)} = \{v \mapsto c\}$.

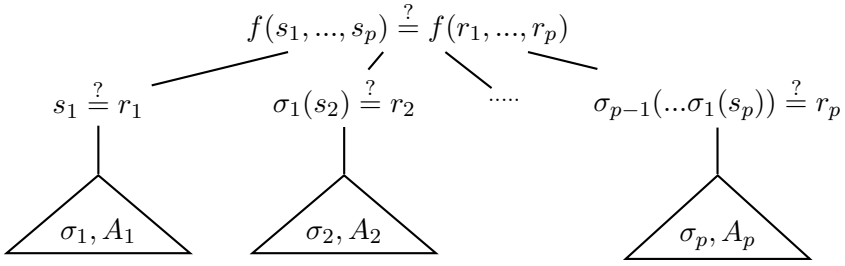
  La propriété est donc vérifiée avec $\theta = \emptyset$.
- $v \stackrel{?}{=} t$, où v est une variable et t un terme.
  La substitution associée est $\{v \mapsto t\}$.
  Or :
  - $\lambda v.v \xrightarrow{comp(v,t)} t$
  - $\sigma_{comp(v,t)} = \{v \mapsto t\}$.

  La propriété est donc vérifiée avec $\theta = \emptyset$.
- $c \stackrel{?}{=} v$, où v est une variable libre et c une constante.
  La substitution associée est $\{v \mapsto c\}$. Notons la θ.
  Nous avons $c = \theta(v)$, donc la suite de transformations vide répond au problème.
  La substitution associée est la substitution vide.
  La propriété est donc vérifiée.

La propriété est donc vérifiée aux feuilles de l'arbre.

4.4.1.2 Preuve : aux nœuds

Décomposition



Supposons que la propriété est vraie pour les différents fils, montrons qu'elle est vraie pour $f(s_1, ..., s_p) \stackrel{?}{=} f(r_1, ..., r_p)$. Nous noterons $[\![A]\!]$ la suite de transformations associée à l'arbre A.

Nous appliquons le même principe que dans la preuve précédente

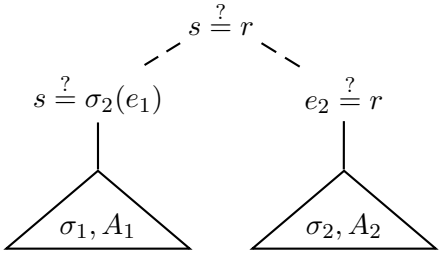
	$f(s_1, ..., s_p)$	
$\xrightarrow{[\![A_1]\!]}$	$\lambda.f(\theta_1(r_1), \sigma_1(s_2), ..., \sigma_1(s_p))$	par hypothèse de récurrence sur le premier fils
$\xrightarrow{[\![A_2]\!]}$	$\lambda.f(\theta_1(r_1), \theta_2(r_2), ..., \sigma_2(\sigma_1(s_p)))$	par hypothèse de récurrence sur le deuxième fils
	...	
$\xrightarrow{[\![A_p]\!]}$	$f(\theta_1(r_1), \theta_2(r_2), ..., \theta_p(r_p))$	par hypothèse de récurrence sur le dernier fils

Puisque les seules variables qui peuvent apparaître dans r sont les variables libres des équations et que les équations sont linéaires par rapport à ces variables, elles n'apparaîtront qu'une fois, donc pour $i \neq j$, $Dom(\theta_i) \cap Dom(\theta_j) = \emptyset$. Soit $\theta = \theta_1 \cup ... \cup \theta_p$, $\theta = \theta_1 \circ ... \circ \theta_p$ car les images des θ_i sont vides.

Nous avons donc :
- $\lambda.f(s_1, ..., s_p) \xrightarrow{[\![A_1]\!];...;[\![A_p]\!]} \lambda.\theta(f(r_1, r_2, ..., r_p))$
- Les différentes substitutions ont des domaines disjoints et des images vides : $(\sigma_1 \circ ... \circ \sigma_p)_{|Var(s)} = (\sigma_{A_1} \circ ... \circ \sigma_{A_p})_{|Var(s)}$

La propriété est donc vraie.

Application d'une équation



Supposons que la propriété est vraie pour les deux fils, montrons qu'elle est vraie pour $s \stackrel{?}{=} r$.

Par hypothèse de récurrence, nous avons : $\lambda.s \xrightarrow{[\![A_1]\!]} \lambda.\theta_1(\sigma_2(e_1))$.

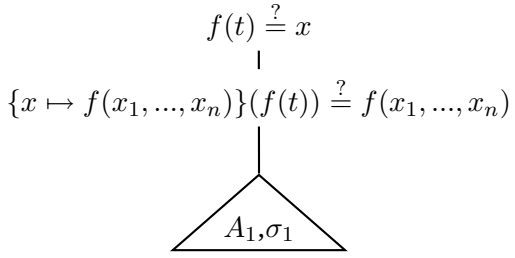
Notons que $Dom(\theta_1) \subset (Var(e_1) \backslash Var(e_2))$ donc $\theta_1(e_2) = e_2$.

	$\lambda.s$	
$\xrightarrow{[\![A_1]\!]}$	$\lambda.\theta_1(\sigma_2(e_1))$	par hypothèse de récurrence sur le problème $s \stackrel{?}{=} \sigma_2(e_1)$
=	$\lambda.\sigma_2(\theta_1(e_1))$	
$\xrightarrow{eq(\epsilon,\theta_1\circ\sigma_2,e_1,e_2)}$	$\lambda.\sigma_2(\theta_1(e_2))$	
=	$\lambda.\sigma_2(e_2)$	
$\xrightarrow{[\![A_2]\!]_E}$	$\theta_2(r)$	par hypothèse de récurrence sur le problème $e_2 \stackrel{?}{=} r$ et en utilisant la propriété 2.5.1 (page 53)

Nous avons donc :
- $\lambda Var(s).s \xrightarrow{[\![A_1]\!];eq(\epsilon,\sigma_2,e_1,e_2);[\![A_2]\!]_E} \theta_2(r)$
- $\sigma_{[\![A_1]\!];eq(\epsilon,\sigma_2,e_1,e_2);[\![A_2]\!]_E} = \sigma_{[\![A_1]\!]}$. Or $\sigma_{1|Var(s)} = \sigma_{[\![A_1]\!]_{|Var(s)}}$ par hypothèse.

La propriété est donc vraie.

Introduction de symboles



Nous avons :

$$\begin{array}{ll} & \lambda.f(t) \\ \xrightarrow{comp(x,f(x_1,...,x_n))} & \lambda.\{x \mapsto f(x_1,...,x_n)\}(f(t)) \\ \xrightarrow{[\![A_1]\!]} & \lambda.\theta(f(x_1,...,x_n)) \end{array}$$

Nous avons donc :

- $\lambda Var(s).f(t) \xrightarrow{comp(x,f(x_1,...,x_n));[\![A_1]\!]} \lambda.\theta(f(x_1,...,x_n))$
- $\sigma_{|Var(f(t))} = (\{x \mapsto f(x_1,...,x_n)\} \circ \sigma_1)_{|Var(f(t))} = \sigma_{1|Var(f(t))}$.

La propriété est donc vraie.

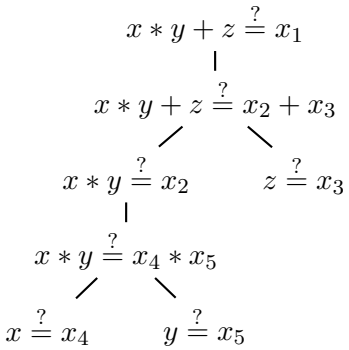
4.4.2 Discussion de la complétude

Nous n'avons pas pour l'instant de preuve de la complétude. Néanmoins, la méthode de construction de cet algorithme, en partant d'un algorithme complet dans un cas particulier et en traitant spécifiquement les éléments ajoutés est raisonnable car elle préserve la complétude du cas précèdent. Nous avons quelques éléments pour émettre la conjoncture que l'algorithme est toujours complet.

Néanmoins, la règle d'introduction d'un symbole (inspirée de **Root Imitation** du système de Gallier et Snyder) n'est peut être pas aussi générale qu'il le faudrait. En effet, les variables intermédiaires ont une forme quelconque. Ici nous choisissons de leur donner une forme identique au terme avec lequel elles sont comparées. Cela limite le nombre de problèmes engendrés à partir d'une telle forme, mais nous perdons peut-être la complétude. Cependant, nous supposons que si nous perdons des solutions, ce sont des solutions moins générales que les solutions trouvées (dans le sens où nous avons une variable libre alors que la solution perdue à une constante contrainte).

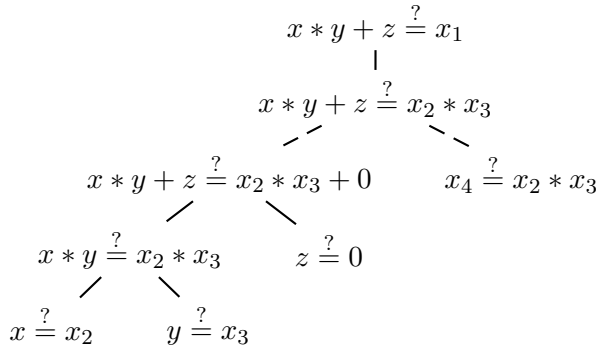
En effet, si nous supposons que la variable libre n'est pas de la forme du terme avec lequel elle est comparée, nous allons essayer de les rendre égaux en appliquant des équations. Nous verrons alors apparaître des contraintes supplémentaires, qui ne sont en fait pas indispensables, puisqu'une solution peut être obtenue sans ces contraintes et avec la forme la plus générale possible.

Exemple 4.4.5
La suite de décomposition suivante, où x, y et z sont des variables du service et x_1 une variable libre :



conduit à la solution très générale : $\{x \mapsto x_4, y \mapsto x_5, z \mapsto x_3\}$.

Par contre, si nous n'imposons aucune forme aux variables libres, nous pouvons par exemple construire :

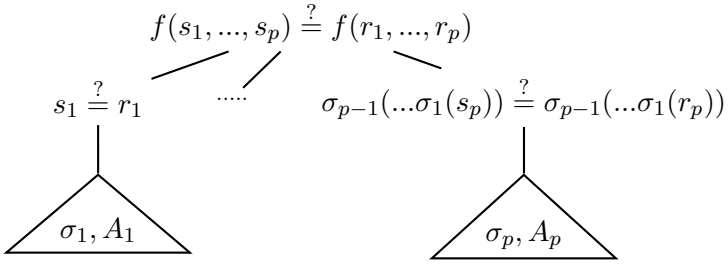


qui engendre la substitution $\{x \mapsto x_2, y \mapsto x_3, z \mapsto 0\}$ *qui est plus contrainte que la précédente.*

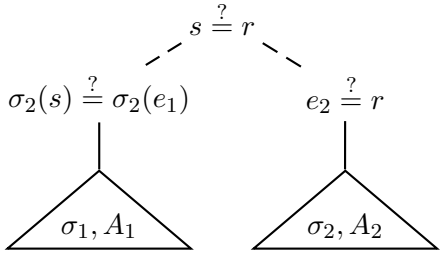
4.5 Étude du cas général

Plaçons nous maintenant dans le cas général ($Var(e_1) \backslash Var(e_2) \neq \emptyset$ et $Var(e_2) \backslash Var(e_1) \neq \emptyset$). Nous devons propager les contraintes obtenues sur les variables libres de e_1.

La décomposition au niveau des frères est donc de la forme :



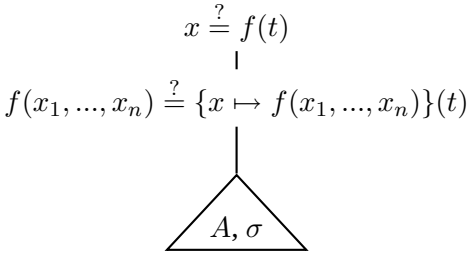
et l'application des équations :



La nouvelle substitution associée à l'application d'une équation n'est plus σ_1 mais $\sigma_2 \circ \sigma_1$ car il faudra propager les contraintes des variables libres du second problème.

Nous perdons alors notre propriété que tous les sous-problèmes sont des problèmes de filtrage équationnel. Nous perdons également la propriété que les problèmes sont des problèmes d'unification équationnelle avec des variables différentes des deux côtés. Par contre, nous savons que si une variable est présente dans les deux membres d'un problème, c'est une variable libre d'une équation.

Nous devons alors ajouter la règle symétrique à celle ajoutée dans le cas précédent :



La substitution solution est $\{x \mapsto f(x_1, ..., x_n)\} \circ \sigma$

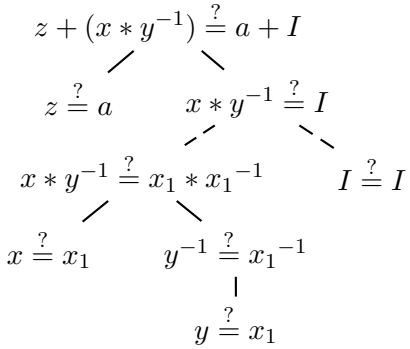
Cette transformation ne peut s'appliquer que dans le cas où x est une variable libre et les x_i sont des nouvelles variables, elles-mêmes considérées comme des variables libres même si elles ne sont pas issues d'une équation.

Les formes résolues sont maintenant celles de la forme $s \stackrel{?}{=} r$, où :

- s est une constante et r est une constante. Si les deux constantes sont différentes, il y a échec. La substitution associée est $\emptyset$.
- s est une constante et r est une variable. La substitution associée est $\{r \mapsto s\}$. Nous sommes ici dans le cas d'une variable libre qui doit être contrainte.
- s est une variable et r est une constante. La substitution associée est $\{s \mapsto r\}$.
- s est une variable et r est une variable. La substitution associée est $\{s \mapsto r\}$.
- s est une variable et r est un terme. La substitution associée est $\{s \mapsto r\}$ si s n'apparaît pas dans le terme, sinon ce n'est pas une forme résolue et il faut appliquer des équations.

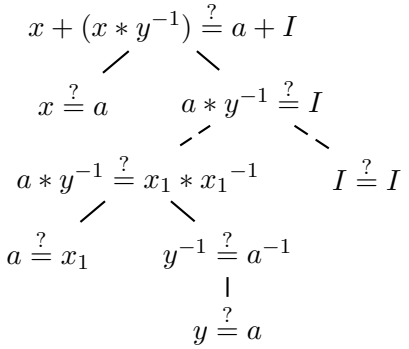
La propriété sur le domaine de σ est maintenant $Dom(\sigma_{|Var(s)\backslash Var(r)}) = Var(s)\backslash Var(r)$. Les variables de s et r pouvant être les mêmes, nous ne sommes pas sûrs que les assignations de deux variables apparaissent effectivement dans la substitution, si ces deux variables sont égales.

Exemple 4.5.1
Exemple simple.

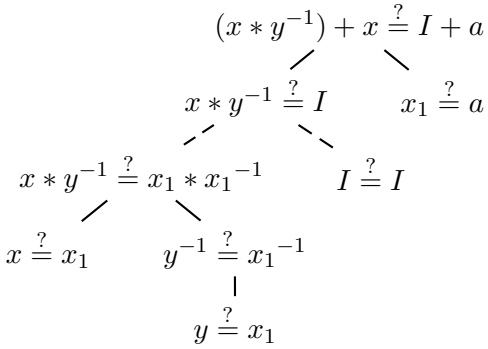


La substitution associée est $\{x \mapsto x_1, y \mapsto x_1, z \mapsto a\}$.

Exemple 4.5.2
Exemple avec propagation des valeurs des x_i.



La substitution associée est $\{x \mapsto a, y \mapsto a, x_1 \mapsto a\}$.



La substitution associée au premier fils est $\sigma_1 = \{x \mapsto x_1, y \mapsto x_1\}$*, celle associée au second fils est* $\sigma_2 = \{x_1 \mapsto a\}$ *donc la substitution solution* $\sigma_1 \circ \sigma_2$ *est* $\{x \mapsto a, y \mapsto a, x_1 \mapsto a\}$.

4.5.1 Preuve de la correction

Soient s et r deux termes tels que $s \stackrel{?}{=} r$ renvoie la solution σ. Montrons qu'il existe une suite de transformations A_E composée uniquement de transformations eq telle que : $\lambda Var(\sigma(s)).\sigma(s) \xrightarrow{A_E} \lambda Var(\sigma(r)).\sigma(r)$.

Nous aurons alors :
$\lambda Var(s).s \xrightarrow{A_{\sigma_{|Var(s)}}} \lambda Var(\sigma(s)).\sigma(s) \xrightarrow{A_E} \lambda Var(\sigma(r)).\sigma(r)$.

Comme dans notre problème initial, r ne possède pas de variable, nous avons alors :
$\lambda Var(s).s \xrightarrow{A_{\sigma_{|Var(s)}}} \lambda Var(\sigma(s)).\sigma(s) \xrightarrow{A_E} r$.

Nous vérifierons la correction de l'algorithme, puisque par construction nous avons l'égalité, sur $Var(s)$, des substitutions.

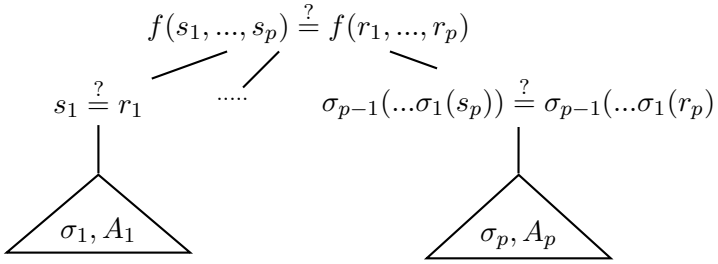
4.5.1.1 Preuve : aux feuilles

- s est une constante et r est une constante.
  Si les deux constantes sont égales, la substitution associée est l'identité. La propriété est trivialement vérifiée.
- s est une constante et r est une variable. La substitution associée est $\{r \mapsto s\}$.
  La propriété est alors vérifiée avec la transformation vide.

– s est une variable et r est une constante. La substitution associée est $\{s \mapsto r\}$.
La propriété est alors vérifiée avec la transformation vide.
– s est une variable et r est une variable. La substitution associée est $\{s \mapsto r\}$.
La propriété est alors vérifiée avec la transformation vide.
– s est une variable et r est un terme. La substitution associée est $\{s \mapsto r\}$ si s n'apparaît pas dans le terme.
La propriété est alors vérifiée avec la transformation vide.

4.5.1.2 Preuve : aux nœuds

Décomposition



Supposons que la propriété est vraie sur les fils, montrons qu'elle est vraie au nœud.

Pour tous les fils, nous avons :

$\lambda.\sigma_i(...\sigma_1(s_i)) \xrightarrow{[\![A_i]\!]} \lambda.\sigma_i(...\sigma_1(r_i))$

D'après la propriété 2.5.4 (page 57), nous avons :

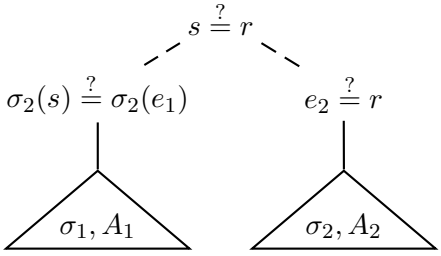
$\lambda.\sigma_n(...\sigma_1(s_i)) \xrightarrow{[\![A_i]\!]} \lambda.\sigma_n(...\sigma_1(r_i))$

Donc :

$\lambda.\sigma_n(...\sigma_1(f(s_1, ..., s_n))) \xrightarrow{[\![A_1]\!];...;[\![A_n]\!]} \lambda.\sigma_n(...\sigma_1(f(r_1, ..., r_n)))$.

La propriété est donc vérifiée.

Application d'équation



Supposons que la propriété est vraie sur les deux fils, montrons qu'elle est vraie au nœud.

Nous avons :

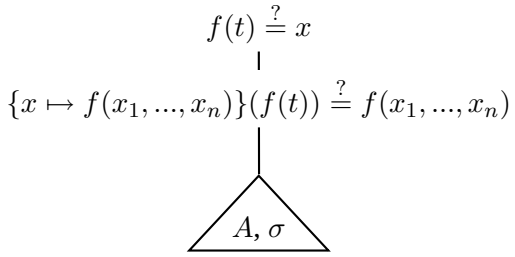
– $\lambda.\sigma_1(\sigma_2(s)) \xrightarrow{[\![A_1]\!]} \lambda.\sigma_1(\sigma_2(e_1))$
– $\lambda.\sigma_2(e_2) \xrightarrow{[\![A_2]\!]} \lambda.\sigma_2(r)$

Donc

$\lambda.\sigma_1(\sigma_2(s)) \xrightarrow{[\![A_1]\!]} \lambda.\sigma_1(\sigma_2(e_1)) \xrightarrow{eq(\epsilon,\sigma_2\circ\sigma_1,e_1,e_2)} \lambda.\sigma_1(\sigma_2(e_2)) \xrightarrow{[\![A_2]\!]} \lambda.\sigma_1(\sigma_2(r))$.

La propriété est donc vérifiée.

Introduction de symboles

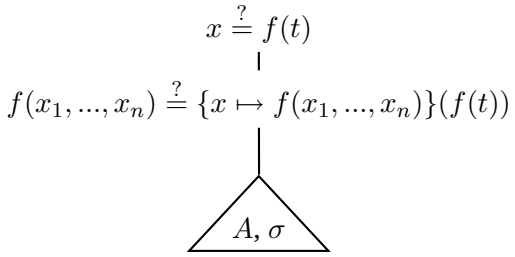


Supposons que la propriété est vraie sur le fils, montrons qu'elle l'est au nœud.

Nous avons :

$\lambda.\sigma(\{x \mapsto f(x_1, ..., x_n)\}(f(t))) \xrightarrow{[\![A]\!]} \lambda.\sigma(\{x \mapsto f(x_1, ..., x_n)\}(x))$

La propriété est donc vérifiée.



Supposons que la propriété est vraie sur le fils, montrons qu'elle l'est au nœud.

Nous avons :

$\lambda.\sigma(\{x \mapsto f(x_1, ..., x_n)\}(x)) \xrightarrow{[\![A]\!]} \lambda.\sigma(\{x \mapsto f(x_1, ..., x_n)\}(f(t)))$

La propriété est donc vérifiée.

4.5.2 Discussion de la complétude

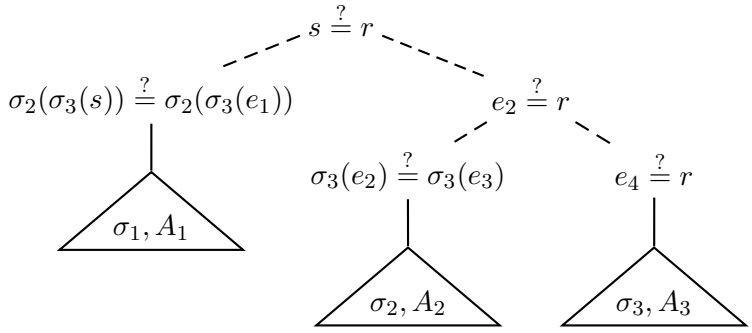
Comme dans le cas précédent, nous avons construit cet algorithme en partant du précédent et en incluant les nouvelles spécificités. Cela nous laisse supposer que nous ne perdons pas de solutions intéressantes, mais nous n'en avons pas réalisé la preuve.

La réalisation de la preuve intégrale de la complétude et des adaptations nécessaires font partie des perspectives futures de nos travaux.

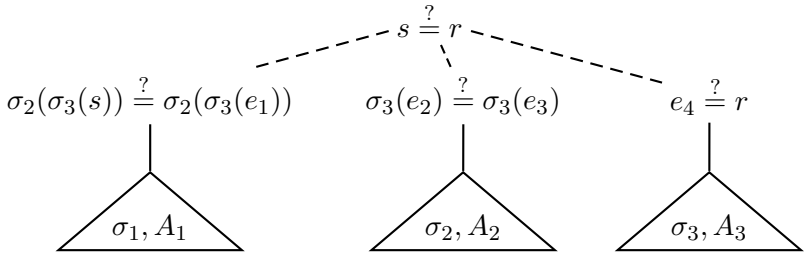
4.6 Traitement des équations

Dans la première version de l'algorithme (traitement en parallèle des sous-problèmes), nous avions modifié la transformation d'application des équations pour ne pas appliquer seulement une équation, mais appliquer une suite d'équations. Cela avait été introduit pour imposer des contraintes sur les symboles à la racine des membres de ces équations. Nous allons faire de même sur cet algorithme.

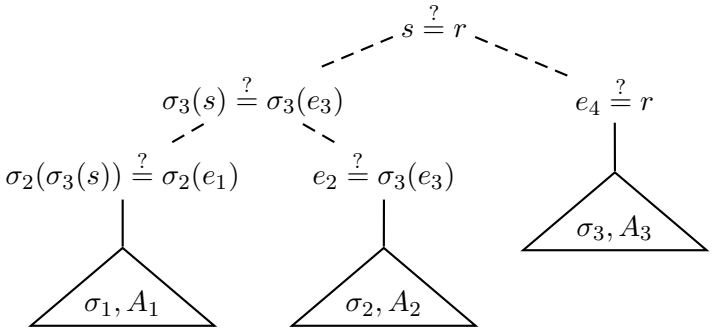
Donc un problème de la forme :



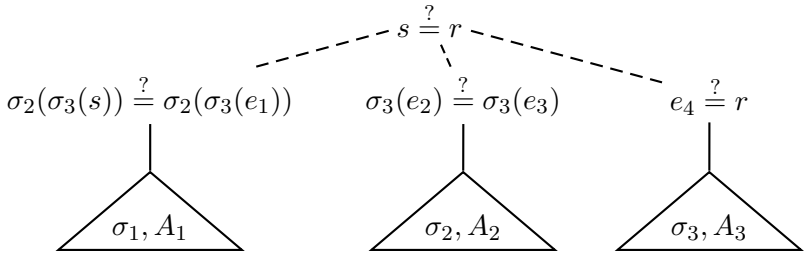
est en fait traité de la façon suivante :



Dans un problème de la forme :

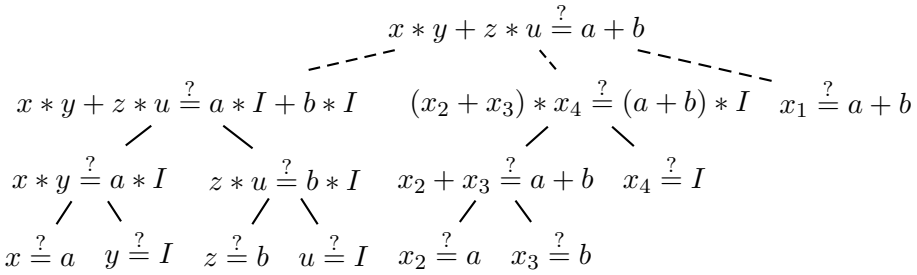


les variables de e_2 et e_1 étant des nouvelles variables, nous avons $\sigma_3(e_2) = e_2$ et $\sigma_3(e_1) = e_1$. Nous traitons également ce problème de la façon suivante :



Il est évident que ces deux méthodes de traitement des équations sont équivalentes. Le second traitement a été choisi, avec interdiction d'appliquer une équation si une décomposition n'a pas été réalisée sur un sous-problème issu de l'application d'une transformation.

Transformation de taille 2 Un service fourni est $s(x, y, z, u) = \lambda x \lambda y \lambda z \lambda u.x * y + z * u$, la requête est $r = a + b$. Parmi les équations se trouvent $v = v * I$ et $(i + j) * k = i * k + j * k$. Une solution est obtenue de la façon suivante :



Solution : $r = s(a, I, b, I)$.

4.7 Complexité

La complexité de l'algorithme est exprimée par rapport au nombre de comparaisons des symboles à la racine des deux termes. Le calcul de la complexité de l'algorithme est fait dans le cas très restrictif où tous les termes sont de profondeur maximale égale à un. Ces termes sont des constantes, des variables ou des termes de la forme $f(x_1, ..., x_n)$, où les x_i sont des constantes ou des variables. Même si cela peut paraître loin de la réalité, cela donne une bonne idée de l'impact des différents paramètres sur l'algorithme.

Dans la suite :
- n_s désigne le nombre de services initiaux ($n_s \neq 0$) ;
- n_{sol} désigne le nombre maximum de solutions obtenues ;
- n_e désigne le nombre d'équations ;
- m désigne le nombre d'équations qui peuvent être appliquées ($n_e \neq 0$ si $m \neq 0$) ;
- d désigne la profondeur maximale de composition ;
- p désigne le nombre maximal de paramètres des services ($p \neq 0$) ;
- a désigne l'arité maximale des opérateurs ($a \neq 0$).

La complexité est calculée dans le pire cas, où toutes les équations peuvent toujours s'appliquer, les services possèdent tous p paramètres, les opérateurs sont tous d'arité a et la profondeur de composition maximale est atteinte sur tous les paramètres des services.

n_{sol} représente le nombre maximum de solutions obtenues en comparant une requête avec tous les services, en supposant que toutes les compositions nécessaires pour que cette solution aboutisse réellement soient possibles. Dans le cas où tous les services sont compatibles avec la requête, cette valeur est liée à n_s (avec toutes les valeurs possibles pour les paramètres). Mais dans le cas général, il ne peut pas être corrélé avec n_s. En effet, le nombre de services peut augmenter sans que n_{sol} augmente, si ces services ne permettent pas de répondre à la requête.

4.7.1 Sans composition

4.7.1.1 Sans équation

Plaçons nous dans le cas où $d = 0$ et $m = 0$. La complexité est :

$$(1 + a) * n_s$$

Pour chaque service, nous comparons les deux symboles à la racine puis chaque fils (au maximum a).

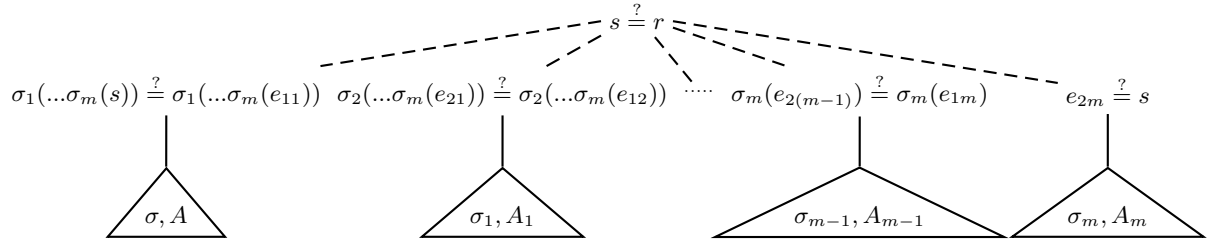
a pouvant être considérée comme une constante, la complexité est :

$$\boxed{\mathcal{O}(n_s)}$$

4.7.1.2 Avec équations

Plaçons nous dans le cas où $d = 0$ et $m \neq 0$. Le nombre de groupements différents de m équations qui peuvent être appliqués est ${n_e}^m$.

Pour chaque service (n_s), nous comparons les symboles à la racine et les fils ($1 + a$) et pour chaque groupement différent (${n_e}^m$), nous engendrons $m+1$ problèmes, sur lesquels nous faisons $1 + a$ comparaisons.



La complexité est donc $((1 + a) + (1 + (1 + a) * (m + 1)) * {n_e}^m) * n_s$.

a pouvant être considérée comme une constante, la complexité est :

$$\boxed{\mathcal{O}(m * {n_e}^m * n_s)}$$

4.7.2 Avec composition

Soit C_d la complexité pour une profondeur de composition égale à d. Nous venons de calculer C_0.

La solution est construite comme si elle ne nécessitait pas de composition. Puis, pour chaque solution obtenue, pour chaque paramètre, l'algorithme peut être exécuté à nouveau avec une profondeur de composition de $d - 1$. Donc :

$C_d = C_0 + (p * n_{sol}) * C_{d-1}$
$= C_0 * (1 + (p * n_{sol}) + (p * n_{sol})^2 + ... + (p * n_{sol})^d)$

Donc :

– si $p * n_{sol} = 1$ (i.e $p = 1$ et $n_{sol} = 1$),

$$\boxed{C_d = \mathcal{O}(C_0 * d)}$$

– si $p * n_{sol} > 1$,

$$\boxed{C_d = \mathcal{O}(C_0 * (p * n_{sol})^d)}$$

4.7.3 Bilan

a pouvant être considérée comme une constante, le tableau suivant résume la complexité de l'algorithme dans les différents cas :

	$m = 0$	$m \neq 0$
$d = 0$	$\mathcal{O}(n_s)$	$\mathcal{O}(m * {n_e}^m * n_s)$
$d \neq 0$ *et* $p * n_{sol} = 1$	$\mathcal{O}(n_s * d)$	$\mathcal{O}(d * m * {n_e}^m * n_s)$
$d \neq 0$ *et* $p * n_{sol} > 1$	$\mathcal{O}(p^d * {n_{sol}}^d * n_s)$	$\mathcal{O}(m * {n_e}^m * p^d * {n_{sol}}^d * n_s)$

Il est important de noter que l'algorithme est :

– linéaire par rapport au nombre de services (si nous supposons que n_{sol} n'est pas corrélé avec n_s) ;
– exponentiel par rapport à la profondeur de composition (choisie par l'utilisateur) ;
– polynomial par rapport au nombre d'équations ;

– exponentiel par rapport au nombre d'équations qui peuvent être appliquées (choisi par l'utilisateur).

Nous avons bien conscience que cette étude est très partielle, mais le problème est très complexe et notre objectif était d'avoir un premier ordre d'idée sur la complexité.

4.8 Implantation de l'algorithme

4.8.1 Le principe de base

4.8.1.1 Résolution de $s \stackrel{?}{=} r$

Le cœur de l'algorithme est la résolution du problème $s \stackrel{?}{=} r$, qui renvoie une liste de substitutions.

L'algorithme est :

Algorithme 2 Algorithme $s \stackrel{?}{=} r$

$compare(s, r, n_{eq})$ =
compare s r **avec**
| $Cste_1, Cste_2 \rightarrow$
si $Cste_1 \neq Cste_2$ **alors**
Échec
| $Var, f(r_1, ..., r_n) \rightarrow$
si $Var \in Var(f(r_1, ..., r_n))$ **alors**
$\forall(\sigma, n'_{eq}) \in compare(f(x_1, ..., x_n), f(r_1, ..., r_n), n_{eq})$
$(\{Var \mapsto f(x_1, ..., x_n)\} \circ \sigma, n'_{eq})$
sinon
$(\{x \mapsto r\}, n_{eq})$
| $Var, r \rightarrow$
$(\{x \mapsto r\}, n_{eq})$
| $f(s_1, ..., s_n)$,$f(r_1, ..., r_n) \rightarrow$
si $n_{eq} > 0$ **alors**
$compareFils(s, r, n_{eq}) \cup compareTransfos(s, r, n_{eq})$
//compareTransfos applique les équations
//compareFils décompose pour comparer les fils
sinon
$compareFils(s, r, n_{eq})$
| $f(s_1, ..., s_n), Var \rightarrow$
si $Var \in Var(f(s_1, ..., s_n))$ **alors**
$\forall(\sigma, n'_{eq}) \in compare(f(s_1, ..., s_n), f(x_1, ..., x_n), n_{eq})$
$(\{Var \mapsto f(x_1, ..., x_n)\} \circ \sigma, n'_{eq})$
sinon
Échec
| $s, r \rightarrow$
si $n_{eq} > 0$ **alors**
$compareTransfos(s, r, n_{eq})$
sinon
Échec
fin compare

Algorithme 3 Algorithme $s \stackrel{?}{=} r$ (sans application d'équation à la racine)

$compareSansTransfo(s, r, n_{eq})$ =
compare s r **avec**
| $Cste_1, Cste_2 \rightarrow$
si $Cste_1 \neq Cste_2$ **alors**
Échec
| $Var, f(r_1, ..., r_n) \rightarrow$
si $Var \in Var(f(r_1, ..., r_n))$ **alors**
$\forall(\sigma, n'_{eq}) \in compare(f(x_1, ..., x_n), f(r_1, ..., r_n), n_{eq})$
$(\{Var \mapsto f(x_1, ..., x_n)\} \circ \sigma, n'_{eq})$
sinon
$(\{x \mapsto r\}, n_{eq})$
| $Var, r \rightarrow$
$(\{x \mapsto r\}, n_{eq})$
| $f(s_1, ..., s_n), f(r_1, ..., r_n) \rightarrow$
$compareFils(s, r, n_{eq})$
| $f(s_1, ..., s_n), Var \rightarrow$
si $Var \in Var(f(s_1, ..., s_n))$ **alors**
$\forall(\sigma, n'_{eq}) \in compare(f(s_1, ..., s_n), f(x_1, ..., x_n), n_{eq})$
$(\{Var \mapsto f(x_1, ..., x_n)\} \circ \sigma, n'_{eq})$
sinon
Échec
| $s, r \rightarrow$
Échec
fin compare

Algorithme 4 Algorithme de comparaison de fils

$compareFils(f(s_1, ..., s_n), f(r_1, ..., r_n), n_{eq})$ =
$l = \{(id, n_{eq})\}$ // ensemble des substitutions générées avec les quantités d'énergie restantes
$l' = \emptyset$ // ensemble des solutions pour un fils
$lsi = \emptyset$ // ensemble des triplets (substitution, quantité d'énergie, nouveau service)
pour $1 \leq i \leq n$ **faire**
// Propagation des contraintes
$\forall(\sigma, n_{eq}) \in l$ ajouter $(\sigma, n_{eq}, \sigma(s_i), \sigma(r_i))$ à lsi
vider l
// Comparaison et calcul des nouvelles contraintes
$\forall(\sigma, n_{eq}, s'_i, r'_i) \in lsi,\ \forall(\sigma', n') \in compare(s'_i, r'_i, n_{eq})$
ajouter $(\sigma \circ \sigma', n')$ à l
fin pour
l

Algorithme 5 Algorithme d'application de la liste des transformations

$compareTransfos(s, r, n_{eq})$ =
$res = \{\}$
pour $1 \leq i \leq n_{eq}$ **faire**
  l =transformations de taille i qui permettent de passer de r à s
  // Application de toutes les transformations
  $\forall t \in l$ ajouter $(compareTransfo(s, r, n_{eq} - i, t, i))$ à res
fin pour
res

Algorithme 6 Algorithme d'application d'une transformation

$compareTransfo(s, r, n_{eq}, t, n)$ =
// Équation avec de nouvelles variables
$t = [(e_{11}, e_{21}); ...; (e_{1n}, e_{2n})]$
// Comparaison de la requête et du premier membre de l'équation
$l = compare(e_{11}, r, n_{eq})$
$l' = \emptyset$
pour $1 \leq i \leq n - 1$ **faire**
  // Calcul des autres sous-problèmes en propageant les contraintes
  $\forall(\sigma, n') \in l,\ \forall(\sigma', n') \in compareSansTransfo(\sigma(e_{1i+1}), \sigma(e_{2i}), n')$ ajouter $(\sigma \circ \sigma', n')$ à l'
  // Nouvel ensemble de contraintes
  $l := l'$
fin pour// Calcul de la solution finale
$\forall(\sigma, n) \in l,\ \forall(\sigma', n') \in compareSansTransfo(s, \sigma(e_{2n}), n)$ ajouter $(\sigma \circ \sigma', n')$ à l'
l'

4.8.1.2 Résolution du problème global

Pour résoudre le problème général, r est comparée avec l'ensemble des services de S comme défini précédemment. Quand, dans une substitution résultat, une variable est associée à un terme, si la profondeur de composition est 0, cette substitution ne fera pas partie de l'ensemble des solutions, sinon ce terme est comparé avec les services de S avec une profondeur de composition décrémentée.

Le nombre d'équations utilisées est propagé en même temps que les contraintes sur les fils. Cela évite le travail redondant présent dans l'algorithme précédent qui effectuait la comparaison pour toutes les valeurs possibles. De plus, cette valeur est aussi connue quand nous relançons pour une solution avec composition. Donc, dans cette version, n_{eq} peut correspondre au nombre maximum d'équations pour obtenir la solution ou au nombre maximum d'équations par relance.

4.8.1.3 Codage

La comparaison de nombres entiers étant moins coûteuse que celle de chaînes de caractères, les variables, les opérateurs et les types sont codés par des entiers.

Nos termes sont du type :

```
type arbre =
```

```
   Vide
 | Cste of (int * int)
 | Var of (int * int)
 | Noeud of noeud
and noeud = {op: int ; fils: arbre  list};;
```

avec :

	< 0	> 0
Var	Variables du service ou des équations	Variables libres
Cste	Constantes de la requête	Constantes du domaine

4.8.1.4 Utilisation de la construction des transformations pour factoriser le travail

Les transformations sont construites comme dans l'algorithme précédent à l'aide d'une fermeture transitive. Nous traitons donc consécutivement des transformations qui ont une partie commune. Pour éviter de refaire le travail, les étapes intermédiaires du calcul sont stockées pour pouvoir reprendre le calcul seulement au moment où les transformations sont différentes.

4.8.2 Options

Les options sont les mêmes que celles du précédent algorithme (section 3.5.4, page 75).

4.8.2.1 Relance sur la requête

Comme dans l'algorithme précédent, il est possible de ne pas relancer le problème pour obtenir une solution à l'aide de compositions dont l'un des composites est la requête. Bien sûr, l'algorithme n'est alors pas complet, mais dans l'optique d'exécuter le service, cette composition ne pourra être que plus complexe qu'une autre solution.

4.8.2.2 Simplification de la requête

Comme dans l'algorithme précédent, il est possible de simplifier la requête. Pour appliquer une règle de réécriture de la forme $e_1 \rightarrow e_2$, le filtre de r et e_1 est calculé et la substitution obtenue est appliquée sur e_2.

4.9 Un mode incrémental ?

Dans cette version de l'algorithme, le nombre d'équations nécessaire pour obtenir une solution étant propagé lors de la construction de cette solution, nous n'avons plus un découpage apparent par nombre d'équations appliquées. Comme de plus les solutions sont propagées sur les sous-problèmes suivants, le traitement des autres sous-problèmes dépend des solutions trouvées précédemment.

Nous pouvons donc moins facilement réutiliser le travail effectué précédemment. Il faudrait stocker tous les problèmes pour lesquels nous avons été bloqués par le manque d'énergie et recalculer toute la suite (partie droite sur les graphiques). Mais cela ne permettra pas de reprendre le travail effectué avec moins d'énergie sur les sous-problèmes suivants.

Le fait de propager les contraintes permet de détecter les erreurs, mais les sous-problèmes ne sont plus les mêmes car ils sont liés aux solutions trouvées sur les autres

sous-problèmes. Cela a des conséquences importantes sur la réutilisation du travail effectué.

4.10 Typage

Nous avons exprimé l'algorithme sans le typage. Il faut également le prendre en compte.

4.10.1 Impact sur l'algorithme

Dans le premier algorithme, la vérification de l'existence d'un type possible pour les variables intermédiaires était retardée à la fin. Dans cet algorithme, les contraintes sur ces variables sont propagées au fur et à mesure, il en est de même pour les types.

Cela a pour conséquences de ne pas avoir un retardement de la détection des erreurs comme dans le premier algorithme, mais une détection de l'erreur dès qu'elle se produit.

4.10.2 Réalisation

L'ajout du typage se résume à ajouter une contrainte pour vérifier que le type de s est supérieur ou égal à celui de r. Si ce n'est pas le cas, il y a un échec.

Il y a deux possibilités pour stocker les équations :

- Dans la matrice de transformations, les opérateurs de même nom mais de types différents ne sont pas regroupés.
  Lors du choix des équations, quand l'opérateur exact du service n'est pas connu (cas d'une équation non typée), tous les opérateurs du même nom et qui ont un type supérieur ou égal à celui de la requête sont choisis.
  Cette solution entraîne une duplication des équations (celles qui sont non typées) et donc une duplication du travail par la suite...
- Dans la matrice de transformations, les opérateurs de même nom sont regroupés.
  Cette solution entraîne l'application d'équations incompatibles pour un problème donné. Mais cette incompatibilité sera vite détectée lors de la décomposition et de la comparaison des fils.

C'est la seconde solution qui a été adoptée pour l'algorithme.

4.11 Interaction entre domaines

Nous venons de décrire comment, dans un domaine donné, grâce à une description appropriée, nous pouvons trouver les services ou les compositions de services qui répondent à notre requête. Les domaines n'étant pas toujours bien cloisonnés, il est intéressant de pouvoir faire une recherche dans plusieurs domaines. Nous verrons dans le chapitre suivant un exemple pratique d'application.

4.11.1 Contraintes sur les domaines

L'interaction entre domaines n'a d'intérêt que si les domaines travaillent sur des éléments communs. En effet, si les domaines sont complètement disjoints, aucune requête ne nécessitera un traitement dans plusieurs de ces domaines. Par contre, s'ils ont des éléments en commun, les termes faisant intervenir ces éléments pourront être traités dans les différents domaines.

Dans ce contexte, les services sont toujours des termes sur la signature d'un seul des domaines, la requête est un terme sur l'union des signatures. Il faut donc que cette union des signatures respecte la contrainte de type des signatures hétérogènes avec sous-typage.

Nous voyons intervenir ici deux notions importantes :
- l'union de signatures qui doit respecter certaines contraintes pour que la requête puisse être valide et qu'il n'y ait pas de conflit dans la définition des éléments communs aux différents domaines ;
- l'intersection des signatures qui ne doit pas être vide pour que l'interaction entre domaines ait un sens.

Nous allons maintenant définir ces deux notions.

Définition 4.11.1 (Union de signatures)
Soient D_1=(S_1,$\leq_1$,Σ_1) et D_2=(S_2,$\leq_2$,Σ_2), deux signatures hétérogènes avec sous-typage, telles que $\leq_{1|S_1 \cap S_2} = \leq_{2|S_1 \cap S_2}$ (les mêmes liens d'héritages sont définis sur les types communs).

Définissons $\leq$ par :
- $\leq_{|S_1 \setminus S_2} = \leq_{1|S_1 \setminus S_2}$
- $\leq_{|S_2 \setminus S_1} = \leq_{2|S_2 \setminus S_1}$
- $\leq_{|S_1 \cap S_2} = \leq_{1|S_1 \cap S_2} = \leq_{2|S_1 \cap S_2}$

Nous en déduisons donc $\leq_{|S_1} = \leq_1$ et $\leq_{|S_2} = \leq_2$.

L'*union des deux signatures* $D_1 \cup D_2$ est ($S_1 \cup S_2$,$\leq$,$\Sigma_1 \cup \Sigma_2$).

Soient D_1=(S_1,$\leq_1$,Σ_1), ..., D_n=(S_n,$\leq_n$,Σ_n), n signatures hétérogènes avec sous-typage. L'union des n signatures est définie par :

$$D_1 \cup D_2 \cup ... \cup D_n = D_1 \cup (D_2 \cup (... \cup D_n))$$

Propriété 4.11.1
L'opérateur $\cup$ sur les signatures est commutatif.

Preuve Évident par symétrie de 1 et 2. □

Propriété 4.11.2
L'opérateur $\cup$ sur les signatures est associatif.

Preuve Soient D_1=(S_1,$\leq_1$,Σ_1), D_2=(S_2,$\leq_2$,Σ_2), D_3=(S_3,$\leq_3$,Σ_3) trois signatures hétérogènes avec sous-typage, telles que :
- $\leq_{1|S_1 \cap S_2} = \leq_{2|S_1 \cap S_2}$
- $\leq_{1|S_1 \cap S_3} = \leq_{3|S_1 \cap S_3}$
- $\leq_{3|S_3 \cap S_2} = \leq_{2|S_3 \cap S_2}$

Définissons $\leq$ par :
- $\leq_{|S_1} = \leq_1$
- $\leq_{|S_2} = \leq_2$
- $\leq_{|S_3} = \leq_3$

Ceci est possible car les opérateurs $\leq_i$ sont égaux sur les parties communes.

Nous avons alors $D_1 \cup (D_2 \cup D_3) = (D_1 \cup D_2) \cup D_3$ =($S_1 \cup S_2 \cup S_3$,$\leq$,$\Sigma_1 \cup \Sigma_2 \cup \Sigma_3$).
□

Définition 4.11.2 (Compatibilité de signatures)
Pour que deux signatures hétérogènes avec sous-typage (S_1,$\leq_1$,Σ_1) et (S_2,$\leq_2$,Σ_2) soient *compatibles*, il faut que :

- les liens d'héritages sur les types en commun soient les mêmes :
$$\leq_{1|S_1\cap S_2} = \leq_{2|S_1\cap S_2}$$
- ($S_1 \cup S_2$,$\leq$,$\Sigma_1 \cup \Sigma_2$) respecte la contrainte de type des signatures hétérogènes avec sous-typage :
$$f \in \left((\Sigma_1 \cup \Sigma_2)_{w_1,s_1} \cap (\Sigma_1 \cup \Sigma_2)_{w_2,s_2}\right) \; et \; w_1 \leq w_2 \Rightarrow s_1 \leq s_2$$
où $\Sigma_{w,s}$ représente les symboles de $w \rightarrow s$, avec $w \in S^*$ et $s \in S$.

- ($S_1 \cup S_2$,$\leq$,$\Sigma_1 \cup \Sigma_2$) respecte les contraintes de typage du $\lambda\&$-calcul :
$$U_i \Downarrow U_j \Rightarrow \exists z \; (unique) \; \in I, \; U_z = inf\{U_i, U_j\}$$

Soient (S_1,$\leq_1$,Σ_1), …, (S_n,$\leq_n$,Σ_n), n signatures hétérogènes avec sous-typage. Ces signatures sont compatibles si elles sont compatibles deux à deux et si l'union des signatures (S,$\leq$,Σ) respecte les contraintes de type :

- $f \in (\Sigma)_{w_1,s_1} \cap (\Sigma)_{w_2,s_2} \; et \; w_1 \leq w_2 \Rightarrow s_1 \leq s_2$
- $U_i \Downarrow U_j \Rightarrow \exists z \; (unique) \; \in I, \; U_z = inf\{U_i, U_j\}$

Définition 4.11.3 (Intersection de signatures)
Soient D_1=(S_1,$\leq_1$,Σ_1) et D_2=(S_2,$\leq_2$,Σ_2), deux signatures hétérogènes avec sous-typage compatibles. L'*intersection de deux signatures* $D_1 \cap D_2$ est ($S_1 \cap S_2$,$\leq_{1|S_1\cap S_2}$,$\Sigma_1 \cap \Sigma_2$).

L'intersection de n domaines (signatures) ne nous intéresse pas. Quand nous travaillons sur plus de deux domaines, il faut que certaines intersections ne soient pas vide, pour que nous n'ayons pas affaire à des domaines totalement séparés.

Quand nous comparons deux termes, les symboles qui nous intéressent particulièrement sont ceux à la racine des termes. Ce sont eux qui déterminent les équations que nous allons appliquer. Ces équations dépendant des domaines, nous avons besoin de déterminer quels sont les domaines des équations qu'il faut appliquer. Pour cela, nous introduisons les notions de « domaines à la racine » et de « domaines d'un problème ».

Définition 4.11.4 (Domaines à la racine)
Soient (S_1,$\leq_1$,Σ_1), …, (S_n,$\leq_n$,Σ_n), n signatures hétérogènes avec sous-typage compatibles, X un ensemble de variables, (S,$\leq$,Σ) l'union des signatures et $t \in T_\Sigma[X]$. Les *domaines à la racine* de t, notés $[t]_d$ sont :

- si $t = c$, où c est une constante, alors $[t]_d$=$\{i|\; c \in \Sigma_i\}$
- si $t = f(p-1, ..., p_n)$, alors $[t]_d$=$\{i|\; f \in \Sigma_i\}$
- si $t = x$, avec $x \in X$ et de type s, alors si s est connu $[t]_d$=$\{i|\; s \in S_i\}$ sinon $\{1, ..., n\}$

Définition 4.11.5 (Domaines d'un problème)
Soient (S_1,$\leq_1$,Σ_1), …, (S_n,$\leq_n$,Σ_n), n signatures hétérogènes avec sous-typage compatibles, X un ensemble de variables, (S,$\leq$,Σ) l'union des signatures et $t_1, t_2 \in T_\Sigma[X]$. Les *domaines d'un problème* $t_1 \stackrel{?}{=} t_2$, notés $[t_1 \stackrel{?}{=} t_2]_d$ sont définis par :

$$[t_1 \stackrel{?}{=} t_2]_d = [t_1]_d \cap [t_2]_d$$

4.11.2 Gestion du multi-domaines

Il y a plusieurs possibilités pour gérer la présence de plusieurs domaines. Ceux-ci peuvent être simplement fusionnés pour qu'ils n'en forment plus qu'un seul. L'union des signatures est alors réalisée comme énoncé précédemment, l'ensemble des équations est l'union des ensembles d'équations, et les services sont l'union des ensembles de services.

L'autre solution est de garder des domaines bien séparés et de travailler dans un domaine ou dans un autre, selon la structure des termes et des problèmes.

Les deux points où les solutions proposées ont un impact sont les suivants :

- L'ensemble des services
  Dans le cas où les domaines sont fusionnés, l'ensemble des services dans lequel la recherche est effectuée est l'union des ensembles de services. Pourtant seuls les services des domaines à la racine de la requête peuvent éventuellement répondre à la requête.
- L'ensemble des équations
  La différence majeure au niveau des équations se trouve au moment de la construction des transformations. En effet, dans le cas où les domaines sont fusionnés pour qu'ils n'en forment plus qu'un, les transformations sont construites en mélangeant des équations des différents domaines. Alors que si les domaines sont bien séparés, les transformations ne sont constituées que d'équations d'un même domaine.

Nous avons choisi la seconde option car elle est, d'une part, moins coûteuse en terme de nombre de comparaisons élémentaires (car moins d'équations sont appliquées) et d'autre part elle respecte plus l'idée de traiter un problème dans le domaine auquel il appartient.

En effet, si nous prenons l'exemple, entièrement développé dans le chapitre suivant, qui fait interagir l'algèbre linéaire et l'optimisation, un domaine regroupant les deux n'aurait pas de signification particulière. Néanmoins, les problèmes d'optimisation manipulant des matrices, il est intéressant de pouvoir avoir accès à l'algèbre linéaire pour faire le traitement sur les matrices avant de résoudre le problème d'optimisation.

4.11.3 Impacts sur l'algorithme

La gestion d'un mode multi-domaines n'a pas un grand impact sur l'algorithme. Les modifications à apporter à l'algorithme se situent au niveau de la vérification des domaines, de la recherche dans une liste de services et de l'application des équations.

- Il faut vérifier que les domaines sont compatibles.
- Quand une requête est traitée de façon globale, c'est-à-dire qu'elle est comparée avec tous les services, les services avec lesquels elle est comparée sont ceux de ses domaines à la racine.
- Les équations qui sont appliquées sur un problème sont celles des domaines du problème traité. Quand l'intersection des domaines des deux parties du problème est nulle, il y a échec de la comparaison.

4.12 Conclusion

Nous venons de présenter dans ce chapitre un algorithme prouvé comme correct, qui permet de répondre au problème de l'utilisateur. Ce second algorithme est plus performant que le premier algorithme proposé grâce à une détection plus précoce des erreurs. Nous avons également prouvé sa complétude pour certaines formes particulières d'équations.

Dans le chapitre suivant, des exemples concrets d'utilisation sont donnés avec une estimation des temps de calcul nécessaires. Dans les exemples donnés, ces temps varient entre une réponse instantanée et quelques minutes.

L'algorithme peut encore être optimisé en ayant un bon mécanisme de réutilisation des problèmes déjà traités et en choisissant mieux l'ordre de traitement des fils dans une décomposition et l'ordre d'application des équations. En effet, les mêmes calculs sont refaits plusieurs fois. Il faut trouver le bon compromis entre le temps passé à chercher si le calcul a déjà été effectué et le temps de refaire ce calcul. Il peut s'avérer plus lent de chercher dans un cache que de refaire le calcul. C'est pourquoi, actuellement dans l'algorithme des mécanismes de cache ont été implémentés seulement à un haut niveau : le traitement global d'une requête, le traitement d'un problème avec le nombre maximum d'équations autorisé. À ces niveaux, les mécanismes de cache sont intéressants car les calculs sont longs.

Chapitre 5

Application à des domaines particuliers

Nous venons de présenter nos propositions pour la description des domaines et des services ainsi que les mécanismes de courtage associés pour répondre aux requêtes de l'utilisateur. Parmi nos contraintes initiales, figurait le fait que les solutions mises en place devrait être générique, c'est-à-dire indépendantes du domaine. Nos propositions valident cette contrainte. Dans ce chapitre, nous allons développer deux domaines d'applications particuliers : l'algèbre linéaire et l'optimisation.

Dans un premier temps, nous verrons l'apport de notre approche dans le cas de l'algèbre linéaire. Nous allons ensuite montrer que ce mécanisme s'adapte également à d'autres domaines. En effet, tout au long du développement, notre domaine de référence a été l'algèbre linéaire mais ce n'est évidemment pas le seul domaine pour lequel le mécanisme peut être utilisé. Nous allons voir ici que nous avons pu décrire le domaine de l'optimisation même si les bénéfices d'une telle approche sont moins importants. En effet, notre mécanisme est intéressant pour l'algèbre linéaire pour deux raisons :

- Les services sont complexes et nous sommes amenés à appliquer les équations dans le sens inverse de la simplification. Par exemple, si un service réalise $\alpha * X * Y$ et que l'utilisateur veut réaliser $A*B$, nous allons appliquer l'équation $x = 1*x$ dans le sens qui introduit 1. Dans un système de réécriture, orienter cette règle dans ce sens entraîne qu'il n'est pas néotherien (ne termine pas). Les outils de réécritures ne sont donc pas adaptés.
- Nous sommes amenés à composer les services.

Nous allons également illustrer la possibilité d'interaction entre les domaines qui a été abordée dans la section 4.11 (page 114). Cette interaction sera développée sur un exemple couplant l'algèbre linéaire et l'optimisation. Mais, nous allons d'abord présenter une interface de saisie des domaines, des services et de la requête.

5.1 Saisie du problème : l'interface web

Nos algorithmes de courtage prennent en entrée : le domaine, les librairies de services et la requête.

Pour faciliter la saisie de ces informations, une interface web a été réalisée. A partir de formulaires jsp, elle permet d'engendrer les fichiers XML contenant les informations nécessaires et respectant les DTD que les algorithmes de courtage attendent en entrée.

5.1.1 Les domaines

Cette interface permet de saisir un nouveau domaine ou d'en sélectionner un existant pour le modifier.

Nous pouvons ajouter, modifier ou supprimer des types, des opérateurs ou des équations et enregistrer ces informations dans un fichier XML respectant la DTD suivante :

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!ELEMENT domain (domainName,sortes,operators,equalities) >
<!ELEMENT domainName (#PCDATA) >
<!ELEMENT sortes (defSorte+) >
<!ELEMENT defSorte (name isSubsorte*) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT isSubsorte (#PCDATA) >
<!ELEMENT operators (operator+) >
<!ELEMENT operator
  (operatorName,parameters,notation,commutativity?,return) >
<!ELEMENT operatorName (#PCDATA) >
<!ELEMENT parameters (parameter*) >
<!ELEMENT parameter (sorte) >
<!ELEMENT sorte (#PCDATA) >
<!ELEMENT notation (#PCDATA) >
<!ELEMENT commutativity (#PCDATA) >
<!ELEMENT return (sorte) >
<!ELEMENT equalities (equality*) >
<!ELEMENT equality (rw,term,term) >
<!ELEMENT rw (#PCDATA) >
<!ELEMENT term (variable | constant | expression) >
<!ELEMENT variable (name,sorte) >
<!ELEMENT constant (value,sorte) >
<!ELEMENT value (#PCDATA) >
<!ELEMENT expression (operatorName,terms) >
<!ELEMENT terms (term+) >
```

Dans l'annexe A.1 (page 149), se trouve un exemple de fichier XML respectant cette DTD et correspondant au domaine simple des exemples. Seules deux équations ont été ajoutées pour ne pas surcharger l'exemple.

Les types sont définis par un nom et la liste de leurs sur-types.

Les opérateurs sont définis par un nom, un champ notation qui indique s'il s'agit d'une notation infixe, préfixe ou postfixe, une indication sur la commutativité de l'opérateur et une signature. Le choix de la signature se fait parmi la liste des types définis, il faut donc définir les types avant les opérateurs.

Les équations sont saisies le plus naturellement possible, par exemple :

```
( a * ( b + c ) ) = ( ( a * b ) + ( a * c ) )
```

Il faut respecter quelques conventions : bien parenthéser et mettre des espaces entre les variables, les opérateurs et les parenthèses. Les noms, les types des variables et une information spécifiant si l'équation doit être orientée pour la simplification des termes, sont précisés séparément.

5.1.2 Les bibliothèques

Une bibliothèque est composée d'un ensemble de services. Il est possible d'ajouter, de modifier ou de supprimer un service et d'enregistrer les modifications dans un fichier XML qui respecte la DTD suivante :

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!ELEMENT library (libraryName,domainName,services) >
<!ELEMENT libraryName (#PCDATA) >
<!ELEMENT domainName (#PCDATA) >
<!ELEMENT services (service+) >
<!ELEMENT service (declaration,specification,cost) >
<!ELEMENT declaration (serviceName,variables,return) >
<!ELEMENT serviceName (#PCDATA) >
<!ELEMENT variables (variable+) >
<!ELEMENT variable (name,sorte) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT sorte (#PCDATA) >
<!ELEMENT return (sorte) >
<!ELEMENT specification (depsType?,(pure|impure|both)) >
<!ELEMENT depsType (depType+) >
<!ELEMENT depType (param1,type,param2,valeur) >
<!ELEMENT param1 (#PCDATA) >
<!ELEMENT type (#PCDATA) >
<!ELEMENT param2 (#PCDATA) >
<!ELEMENT valeur (#PCDATA) >
<!ELEMENT pure (term) >
<!ELEMENT term (variable|constant|expression|match) >
<!ELEMENT variable (name,sorte) >
<!ELEMENT constant (value,sorte) >
<!ELEMENT value (#PCDATA) >
<!ELEMENT expression (operatorName,terms) >
<!ELEMENT operatorName (#PCDATA) >
<!ELEMENT terms (term+) >
<!ELEMENT match (case+) >
<!ELEMENT case (variable,constant,term) >
<!ELEMENT impure (inout+) >
<!ELEMENT inout (variable,term) >
<!ELEMENT both (term,inout+)>
```

Dans l'annexe A.2 (page 152), se trouve un exemple de fichier XML respectant cette DTD et correspondant à une librairie contenant les services :

- $s_1 = \lambda x \lambda y \lambda z.x + (y + z)$
- $s_2 = \lambda x \lambda y.x * y$
- $s_3 = \lambda x \lambda y \lambda z.x * (y + z)$

Chaque service est saisi en donnant son nom, ses paramètres (noms et types) et la fonctionnalité qu'il réalise. Cette fonctionnalité est saisie de la même façon que les équations, en ajoutant les « dépendances de type », qui permettent de lier le type réel d'un paramètre avec la valeur d'un autre paramètre.

La fonctionnalité d'un terme pourra être plus complexe qu'un simple terme de l'algèbre. Nous ajoutons un « match » qui permet de définir des fonctionnalités différentes en fonction de la valeur d'un paramètre.

Exemple 5.1.1
Voici un exemple de dépendances de type. Dans le BLAS, la procédure $strsm$*, dont la signature est :*
$strsm(side : Char, uplo : Char, transA : Char, diag : Char, m : Int, n : Int, \alpha : Real, A : TriInvMatrix, lda : Int, B : Matrix, ldb : Int)$.
résout un système linéaire quand la matrice A *est triangulaire.*

Si cette matrice est triangulaire supérieure, le paramètre $uplo$ *devra être positionné à* $'u'$ *sinon, il vaudra* $'l'$*. Si la matrice* A *est unitaire, le paramètre* $diag$ *devra être positionné à* $'u'$*, sinon il vaudra* $'n'$*. Cela sera exprimé par une « dépendance de types ».*

De plus, la fonctionnalité rendue est différente selon la valeur du paramètre $side$*. Nous exprimons le service réalisé par :*
$B \leftarrow match$
$\quad | \ (side,' l') \rightarrow (\alpha * (op(transA, (A^{-1})) * B))$
$\quad | \ (side,' r') \rightarrow (\alpha * (B * op(transA, (A^{-}1))))$

5.1.3 Les requêtes

Les requêtes sont définies en précisant les constantes de la requête et la fonctionnalité recherchée. Cette fonctionnalité est saisie comme les équations, de façon naturelle, en respectant les quelques règles énoncées précédemment. Elles sont également enregistrées dans un fichier XML qui respecte une DTD particulière.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!ELEMENT requete (domainName,term) >
<!ELEMENT domainName (#PCDATA) >
<!ELEMENT term (variable|constant|expression) >
<!ELEMENT variable (name,sorte) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT sorte (#PCDATA) >
<!ELEMENT constant (value,sorte) >
<!ELEMENT value (#PCDATA) >
<!ELEMENT expression (operatorName,terms) >
<!ELEMENT operatorName (#PCDATA) >
<!ELEMENT terms (term+) >
```

Dans l'annexe A.3 (page 156), se trouve un exemple de fichier XML respectant cette DTD et correspondant à la requête $a * b + c * d$.

5.1.4 Exécution

L'interface permet également d'exécuter l'algorithme. Il faut choisir :
- le(s) domaine(s) de l'application ;
- les librairies dans lesquelles il faut chercher (parmi les librairies définies dans ce domaine) ;
- la requête ;

– des entiers qui déterminent la quantité d'énergie nécessaire.

Les solutions au problème sont alors affichées dans le navigateur. Cela est réalisé grâce à la procédure Java *exec* de la classe *runtime*.

5.2 L'algèbre linéaire

Après avoir expliqué les raisons du choix de l'algèbre linéaire, sa représentation sous forme de spécification algébrique sera développée. Puis nous étudierons deux librairies et la représentation de leurs services dans notre formalisme. Enfin, nous développerons des exemples pour illustrer les possibilités du mécanisme de courtage mis en place. Toutes les données relatives aux quantités d'énergie et aux temps de calcul seront données pour le second algorithme de courtage.

5.2.1 Pourquoi ce domaine ?

L'algèbre linéaire est le domaine qui nous a particulièrement intéressé. En effet, ce travail s'est déroulé au sein du projet GRID-TLSE [1] [CDL+06, CDD+05] qui a pour but de créer un site d'expertise en algèbre linéaire creuse. L'interaction forte avec la communauté de l'algèbre linéaire a facilité l'étude de ce domaine. Toutes les étapes de développement ont donc d'abord été validées sur ce domaine.

5.2.1.1 Le projet TLSE

Le projet TLSE vise à construire un site d'expertise pour la résolution de systèmes linéaires creux par méthodes directes. De nombreux outils ont été développés dans ce cadre. Ces outils ont le même objectif global (généralement résoudre un système linéaire de grande taille) mais utilisent des algorithmes différents ou des variantes d'un même algorithme. Le site d'expertise doit, d'une part, aider un utilisateur dans le choix de l'outil le plus adapté à son problème et d'autre part, lui proposer des valeurs pertinentes pour les paramètres de contrôle de l'outil retenu selon son problème.

Pour cela, le site (WEBSOLVE) demande à l'utilisateur de décrire précisément son problème ainsi que les caractéristiques des machines et des outils qui l'intéressent.

Le moteur d'expertise (WEAVER) exploite ensuite les informations fournies, les scénarios d'expertise définis par les experts du domaine ainsi que les caractéristiques des machines et outils disponibles pour construire des plans d'expériences qui seront exécutés sur la grille en exploitant des intergiciels adéquats (TLSE exploite l'intergiciel DIET développé dans le cadre du projet GRID-ASP par F. Desprez et al. [CDL+02]). L'ensemble des résultats et mesures est ensuite utilisé pour construire des diagrammes synthétiques présentés à l'utilisateur pour l'aider dans le choix des machines et outils adaptés à ses besoins. Les valeurs des paramètres de contrôle lui sont également accessibles pour l'aider dans l'exploitation optimale des outils.

D'autre part, ce site doit permettre aux experts de disposer d'un atelier de comparaison des différents outils disponibles. Il doit pour cela offrir un accès aux différents outils de manière uniforme.

Le site propose également une bibliothèque de problèmes standard ainsi que des références bibliographiques.

[1] `http://www.enseeiht.fr/lima/tlse/`

5.2.1.2 Les spécificités des bibliothèques

Les bibliothèques d'algèbre linéaire ne sont pas utilisées que par des spécialistes du domaine et ne sont pas toujours facilement exploitables. Les méthodes présentent des signatures importantes et réalisent des services complexes. En effet, dans un souci d'optimisation, il peut être plus intéressant de construire algorithmiquement la composition de services élémentaires plutôt qu'exécuter les services élémentaires en séquence, dans le but de minimiser les ressources utilisées. Pour réaliser l'addition de deux matrices avec le BLAS, il faudra par exemple utiliser une procédure qui réalise $\alpha * A * B + \beta * C$ (voir la description du BLAS en annexe B.4.3, page 165). De plus, ces algorithmes sont également optimisés pour des propriétés particulières de leurs paramètres. Selon les propriétés de ces paramètres, un algorithme est préféré à un autre.

5.2.2 Formalisation du domaine

L'algèbre linéaire est un domaine qui se prête bien à une formalisation sous forme de spécification algébrique. En effet, elle comporte quatre types de base représentant les caractères, les scalaires, les vecteurs et les matrices, qui peuvent être déclinés selon les propriétés des éléments (réel, complexe, symétrique, inversible, ...). Les opérateurs sont ceux représentant les opérations qui peuvent être réalisées sur ces éléments : addition, multiplication, transposition, ... Les propriétés de ces éléments (associativité, distributivité, éléments neutres, ...) sont bien connues.

5.2.2.1 Les types

Pour les descriptions qui vont suivre, nous nous plaçons dans l'espace des réels $\mathbb{R}$, mais toutes les descriptions peuvent être très facilement étendues au cas complexe.

Les types choisis pour représenter l'algèbre linéaire sont ceux décrivant :

- les caractères
  - Char
- les scalaires
  - Int, NzInt
  - Real, NzReal
- les vecteurs
  - Vector
  - VectorL (vecteurs lignes)
- les matrices
  - Matrix
  - InvMatrix (matrices inversibles)
  - SymMatrix (matrices symétriques)
  - ...

L'annexe B.1 (page 159) contient l'ensemble des types définis.

Ces différents types ont des liens de compatibilité représentés dans les figures 5.1 et 5.2.

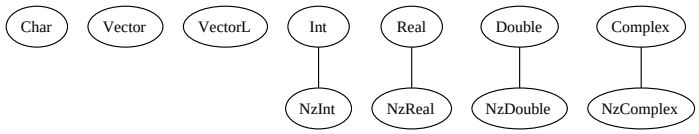


FIG. 5.1 – Liens d'héritage entre les différents types de l'algèbre linéaire

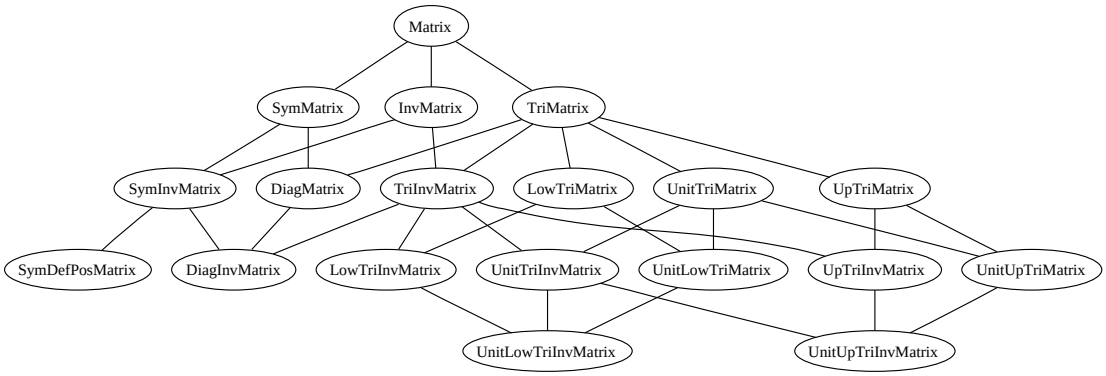


FIG. 5.2 – Liens d'héritage entre les différents types de l'algèbre linéaire

5.2.2.2 Les opérateurs

Les opérateurs choisis sont ceux représentant l'addition, la soustraction, la multiplication, l'inverse, la transposé, les factorisations, ... Ils peuvent être déclarés avec plusieurs signatures différentes. Ces signatures permettent également d'exprimer la conservation de certaines propriétés par un opérateur. Par exemple, la conservation de la symétrie d'une matrice par la transposition pourra être exprimée. « [comm] » signifie que l'opérateur est commutatif.

Les opérateurs représentant :

- l'addition :
  - + :Matrix $\times$ Matrix $\rightarrow$ Matrix [comm]
  - + :SymMatrix $\times$ SymMatrix $\rightarrow$ SymMatrix [comm]
  - ...
- la soustraction :
  - - : Matrix $\times$ Matrix$\rightarrow$ Matrix
  - ...
- la multiplication :
  - * :Matrix $\times$ Matrix $\rightarrow$ Matrix
  - * :InvMatrix $\times$ InvMatrix $\rightarrow$ InvMatrix
  - * :Real $\times$ Matrix $\rightarrow$ Matrix
  - ...
- l'inverse :
  - $\hat{}$-1 :InvMatrix $\rightarrow$ InvMatrix
  - ...
- la transposée :

  - $\hat{}$T :Matrix → Matrix
  - $\hat{}$T :TriMatrix → TriMatrix
  - ...
- les factorisations :
  - choleskyU :SymDefPosMatrix → UpTriInvMatrix
  - choleskyL :SymDefPosMatrix → LowTriInvMatrix
  - l :InvMatrix → UnitLowTriInvMatrix
  - u :InvMatrix → UpTriInvMatrix
  - pivlu :InvMatrix → Vector
- les normes :
  - norm1 :Vector → Real
  - norm2 :Vector → Real
- ...

Les constantes choisies sont celles représentant les éléments neutres et les éléments absorbants des opérateurs ci-dessus, ainsi que quelques caractères :

- 1 : → NzInt
- 0 : → Int
- 1.0 : → NzReal
- 0.0 : → Real
- I : → DiagInvMatrix
- O : → Matrix
- 'n' : → Char
- ...

L'ensemble des opérateurs et des constantes est consultable dans l'annexe B.2 (page 159).

5.2.2.3 Les équations

Les propriétés des opérateurs sont exprimées sous forme d'équations. Lorsque le type d'une variable n'est pas précisé, cela signifie que l'équation est valide quelque soit le type de cette variable, si les deux termes sont bien typés.

« => » signifie que cette règle peut être utilisée de manière orientée avec l'option de simplification d'une requête. « == » signifie que l'équation n'est jamais orientée.

Les propriétés définies dans le cas de l'algèbre linéaire sont les suivantes :

- éléments neutres
  - => $(1.0 * a) = a$
  - => $(a + O) = a$
  - ...
- éléments absorbants
  - => $(a * O) = O$
  - ...
- distribution / factorisation
  - == $((a * b)\hat{} - 1) = ((b\hat{} - 1) * (a\hat{} - 1))$
  - == $((a\hat{}T)\hat{} - 1) = ((a\hat{} - 1)\hat{}T)$
  - ...
- associativité
  - == $(a * (b * c)) = ((a * b) * c)$
  - == $(a + (b + c)) = ((a + b) + c)$
- ...

L'ensemble des équations est présenté dans l'annexe B.3 (page 162).

5.2.3 Les bibliothèques

Parmi les bibliothèques d'algèbre linéaire, nous en avons choisi deux particulièrement utilisées : BLAS et LAPACK. Ces deux bibliothèques ont été créées avec un souci de performance, nous y trouvons donc des services complexes comme évoqués précédemment.

5.2.3.1 BLAS

Le BLAS (Basic Linear Algebra Subprograms) [BDD+02, DDDH90] regroupe des procédures qui réalisent des opérations sur les matrices et vecteurs de façon performante. Le niveau 1 du BLAS s'intéresse aux opérations scalaire - vecteur et vecteur - vecteur. Le niveau 2 s'intéresse aux opérations matrice - vecteur et le niveau 3 aux opérations matrice - matrice. Le BLAS, étant efficace et portable, est couramment utilisé dans d'autres librairies, comme c'est le cas pour LAPACK, qui propose des opérations de plus haut niveau.

Niveau 1 Parmi les procédures de niveau 1, se trouvent les additions de vecteurs, les multiplications par un scalaire, le produit scalaire et les calculs de normes. Dans notre formalisme, les procédures du niveau 1 du BLAS s'expriment de la façon suivante :

- $sscal(n : Int, \alpha : Real, x : Vector, incx : Int)$:
  $x \leftarrow (\alpha * x)$
- $saxpy(n : Int, \alpha : Real, x : Vector, incx : Int, y : Vector, incy : Int)$:
  $y \leftarrow ((\alpha * x) + y)$
- ...

Les procédures du niveau 1 du BLAS que nous avons décrites sont disponibles en annexe B.4.1 (page 164).

Niveau 2 Parmi les procédures de niveau 2, se trouvent les multiplications matrice - vecteur, les résolutions de systèmes d'équations, ... Dans notre formalisme, les procédures du niveau 2 du BLAS s'expriment de la façon suivante :

- $sgemv(trans : Char, m : Int, n : Int, \alpha : Real, a : Matrix, lda : Int, x : Vector, incx : Int, \beta : Real, y : Vector, incy : Int)$:
  $y \leftarrow match$
  $\mid (trans,' n') \rightarrow ((\alpha * (a * x)) + (\beta * y))$
  $\mid (trans,' t') \rightarrow ((\alpha * ((a\hat{}T) * x)) + (\beta * y))$
- $ssymv(uplo : Char, n : Int, \alpha : Real, a : Matrix, lda : Int, x : Vector, incx : Int, \beta : Real, y : Vector, incy : Vector)$:
  $y \leftarrow ((\alpha * (a * x)) + (\beta * y))$
- ...

Les procédures du niveau 2 du BLAS que nous avons décrites sont disponibles en annexe B.4.2 (page 164).

Niveau 3 Parmi les procédures de niveau 3, se trouvent les additions et multiplications de matrices, le calcul de la transposée et la résolution de systèmes linéaires. Dans notre formalisme, les procédures du niveau 3 du BLAS s'expriment de la façon suivante :

- $sgemm(transa : Char, transb : Char, m : Int, n : Int, k : Int, \alpha : Real, A : Matrix, lda : Int, B : Matrix, ldb : Int, \beta : Real, C : Matrix, ldc : Int)$:

$Matrix$
$C \leftarrow ((\alpha * (op(transa, A) * op(transb, B))) + (\beta * C))$

- $strsm(side : Char, uplo : Char, transA : Char, diag : Char, m : Int, n : Int, \alpha : Real, A : TriInvMatrix, lda : Int, B : Matrix, ldb : Int) : void$
  $[(A, UpTriInvMatrix, uplo,' u'), (A, LowTriInvMatrix, uplo,' l'), (A, UnitTriInvMatrix, diag,' u'), (A, Matrix, diag,' n')]$
  $B \leftarrow match$
  $\mid (side,' l') \rightarrow (\alpha * (op(transA, (A\hat{} - 1)) * B))$
  $\mid (side,' r') \rightarrow (\alpha * (B * op(transA, (A\hat{} - 1))))$
- ...

Les procédures du niveau 3 du BLAS que nous avons décrites sont disponibles en annexe B.4.3 (page 165).

5.2.3.2 LAPACK

LAPACK [ABB+99] (Linear Algebra PACKage) est une bibliothèque d'algèbre linéaire écrite en Fortran77 qui contient des procédures pour résoudre les systèmes linéaires, le calcul des valeurs propres et des valeurs singulières. Les factorisations de matrices (LU, Cholesky, ...) sont aussi fournies. Les matrices denses et par bandes sont traitées, mais pas les matrices creuses. Dans tous les cas, les mêmes fonctionnalités sont fournies pour des matrices réelles ou complexes et en simple ou double précision.

LAPACK possédant un nombre très important de procédures (plus d'une centaine de procédures dans le cas des réels en simple précision), nous n'avons décrit qu'une petite partie de LAPACK dans notre formalisme :

- $spotrf(uplo : Char, A : SymDefPosMatrix, info : Int) : Void\ A \leftarrow match$
  $\mid (uplo,' u') \rightarrow choleskyU(A)$
  $\mid (uplo,' l') \rightarrow choleskyL(A)$
- $slaswp(n : Int, A : Matrix, lda : Int, k1 : Int, k2 : Int, ipiv : Vector, incx : Int) : Void$
  $A \leftarrow perm(ipiv, A)$
- $sgetrf(m : Int, n : Int, A : InvMatrix, lda : Int, ipiv : Vector, info : Int) : Void$
  $A \leftarrow l(A)$
  $A \leftarrow u(A)$
  $ipiv \leftarrow pivlu(A)$

Nous remarquons dans ce dernier exemple que la matrice A est affectée à deux valeurs différentes. Nous avons accepté ce qui peut sembler être une erreur, pour répondre au fait qu'un même objet représente en fait deux objets différents. En effet la matrice A représente dans sa partie supérieure, la matrice U (issue de la factorisation LU) et, dans sa partie inférieure, la matrice L.

5.2.4 Des exemples

Nous allons maintenant développer des exemples d'utilisation de l'algorithme de courtage. Nous nous positionnons dans l'algèbre linéaire dense avec les descriptions du domaine et des services développés précédemment.

Dans tous les exemples, les résultats donnés sont prélevés parmi tous les résultats renvoyés par l'algorithme. Le nombre d'équations autorisées, la taille maximale des transformations et la profondeur de composition donnés sont les valeurs minimales nécessaires pour les obtenir. Si l'une de ces valeurs est augmentée, le nombre de résultats augmentera de même que le temps de calcul.

Les temps donnés sont ceux évalués :

- en calculant le temps CPU grâce à la commande time de Linux ;
- en utilisant les valeurs minimales des paramètres ;
- en interdisant les relances sur la requête ;
- en simplifiant la requête ;
- en utilisant une machine :
  - Intel(R) Pentium(R) IV CPU 1.80GHz
  - RAM : 512 MB
  - taille du cache : 512 KB

L'algorithme utilisé est le second dans sa version la plus élaborée. Néanmoins, pour qu'il soit plus efficace, les variables libres ont été remplacées par des constantes (puisque pour invoquer le service, l'utilisateur leur donnera une valeur particulière). Nous n'avons donc plus de variables libres et l'algorithme équivaut alors au second algorithme dans sa version simple. Cela entraîne un gain de temps, car moins de sous-problèmes sont engendrés (puisque les variables libres seront contraintes aux constantes au lieu de pouvoir prendre toutes les formes et tous les types possibles). Nous n'utilisons donc pas la règle d'introduction des symboles.

Les exemples présentés dans la suite sont ceux exposés dans [DHP06].

5.2.4.1 Exemple 1

Les services disponibles sont ceux du niveau 3 du BLAS. La requête de l'utilisateur est $A,\ B,\ C : Matrix \quad A * (B * C)$.

Une combinaison de services générée par l'algorithme est :

```
p1=0;
sgemm('n', 'n', ?, ?, ?, 1.0, B, ?, C, ?, 1.0, p1, ? );
  //p1<-B*C
p2=0;
sgemm('n', 'n', ?, ?, ?, 1.0, A, ?, p1, ?, 1.0, p2, ? );
  //p2<-A*p1
p2;
```

où les paramètres affectés à « ? » sont ceux qui ne peuvent pas être déterminés.
Pour trouver cette solution, l'algorithme doit être paramétré avec :

- nombre d'équations autorisées : 5
- taille maximale des transformations : 1
- profondeur de composition : 1

La solution est obtenue **instantanément**. L'algorithme trouve **8** autres solutions.

5.2.4.2 Exemple 2

Les services disponibles sont ceux du niveau 3 du BLAS et ceux de LAPACK. L'utilisateur veut résoudre le système linéaire $Ax = B$ (où aucune propriété n'est connue sur A). Une réponse calculée par l'algorithme est :

```
p1=A;
p2=?;
sgetrf(?, ?, p1, ?, p2, ? );
  //p2<-fatorisation LU de A (A= P*L*U)
p3=B;
slaswp(?, p3, ?, ?, ?, p2, ? );
  //p3<-permutation des lignes de B
p4= p3;
strsm('l', 'l', 'n', 'n', ?, ?, 1.0, p1, ?, p4, ? );
  //résout L*x=p3; p4<-x;
p5= p4;
strsm('l', 'u', 'n', 'n', ?, ?, 1.0, p1, ?, p5, ? );
  //résout U*x=p4; p5<-x;
p5;
```

Pour trouver cette solution, l'algorithme doit être paramétré avec :

- nombre d'équations autorisées : 6
- taille maximale des transformations : 2
- profondeur de composition : 3

La solution est obtenue en **2m70s**. L'algorithme ne trouve aucune autre solution.

5.2.4.3 Exemple 3

Cet exemple est le même que le précédent, à la différence près que la matrice A est une matrice symétrique définie positive.

L'algorithme retourne ces solutions :

```
p1=A;
p2=?;
sgetrf(?, ?, p1, ?, p2, ? );
  //p2<-fatorisation LU de A (A= P*L*U)
p3=B;
slaswp(?, p3, ?, ?, ?, p2, ? );
  //p3<-permutation des lignes de B
p4= p3;
strsm('l', 'l', 'n', 'n', ?, ?, 1.0, p1, ?, p4, ? );
  //résout L*x=p3; p4<-x;
p5= p4;
strsm('l', 'u', 'n', 'n', ?, ?, 1.0, p1, ?, p5, ? );
  //résout U*x=p4; p5<-x;
p5;
```

Pour trouver cette solution, l'algorithme doit être paramétré avec :

- nombre d'équations autorisées : 6
- taille maximale des transformations : 2
- profondeur de composition : 3

La solution est obtenue en **3m20s**. L'algorithme trouve **2** autres solutions.

Une autre solution trouvée est :

```
p1=A;
spotrf('u', p1, ?, );
  //p1<- factorisation de Cholesky de A (A=U{^T}*U)
p2=B;
strsm('l', 'u', 't', 'n', ?, ?, 1.0, p1, ?, p2, ? );
  //résout U{^T}*x=B; p2<-x;
p3= p2;
strsm('l', 'u', 'n', 'n', ?, ?, 1.0, p1, ?, p3, ? );
  //résout U*x=p2; p3<-x;
p3;
```

Pour trouver cette solution, l'algorithme doit être paramétré avec :

- nombre d'équations autorisées : 5
- taille maximale des transformations : 1
- profondeur de composition : 3

La solution est obtenue **instantanément**. L'algorithme ne trouve pas d'autre solution.

La solution suivante est également renvoyée :

```
p1=A;
spotrf('l', p1, ? );
  //p1<- factorisation de Cholesky de A (A=L*L{^T})
p2=B;
strsm('l', 'l', 'n', 'n', ?, ?, 1.0, p1, ?, p2, ? );
  //résout L*x=B; p2<-x;
p3= p2;
strsm('l', 'l', 't', 'n', ?, ?, 1.0, p1, ?, p3, ? );
  //résout L{^T}*x=p2; p3<-x;
p3;
```

Pour trouver cette solution, l'algorithme doit être paramétré avec :

- nombre d'équations autorisées : 6
- taille maximale des transformations : 2
- profondeur de composition : 3

La solution est obtenue en **3m20s**. L'algorithme trouve **2** autres solutions.

La première solution est la même que dans le cas général. Les autres utilisent le fait que A est une matrice symétrique définie positive et remplace la factorisation LU par une factorisation de Cholesky, ce qui est une meilleure solution en terme de complexité et donc de temps de calcul.

5.2.4.4 Exemple 4

L'exemple est le même que les précédents mais avec A une matrice inversible triangulaire supérieure.

La solution suivante est trouvée :

```
p1=B;
strsm('l', 'u', 'n', 'n', ?, ?, 1.0, A, ?, p1, ? );
  //résout A*x=B; p1<-x;
p1;
```

Pour trouver cette solution, l'algorithme doit être paramétré avec :

- nombre d'équations autorisées : 2
- taille maximale des transformations : 1
- profondeur de composition : 0

La solution est obtenue **instantanément**. L'algorithme ne trouve pas d'autre solution.

5.2.4.5 Bilan

	Exemple 1	Exemple 2	Exemple 3	Exemple 4
Nombre d'équations autorisées	2	2	2	2
Taille maximale des transformations	1	1	1	1
Profondeur de composition	0	0	0	0
Nombre de solutions	**0**	**0**	**0**	**1**
Temps de calcul	**Inst.**	**Inst.**	**Inst.**	**Inst.**
Nombre d'équations autorisées	5	5	5	5
Taille maximale des transformations	1	1	1	1
Profondeur de composition	1	1	1	1
Nombre de solutions	**9**	**0**	**0**	**1**
Temps de calcul	**Inst.**	**Inst.**	**Inst.**	**Inst.**
Nombre d'équations autorisées	5	5	5	5
Taille maximale des transformations	1	1	1	1
Profondeur de composition	3	3	3	3
Nombre de solutions	**9**	**0**	**1**	**1**
Temps de calcul	**Inst.**	**Inst.**	**Inst.**	**Inst.**
Nombre d'équations autorisées	6	6	6	6
Taille maximale des transformations	2	2	2	2
Profondeur de composition	3	3	3	3
Nombre de solutions	**108**	**1**	**3**	**2**
Temps de calcul	**40s**	**2m70s**	**3m20s**	**2m80s**

Notons que les solutions qui demandent actuellement 3m20s pour être obtenues, ne pouvaient pas être obtenues avant que des optimisations soient réalisées sur le code des algorithmes. En effet, les temps de calcul étaient très longs et la mémoire nécessaire trop importante par rapport à ce dont disposait l'ordinateur sur lequel étaient lancés ces exemples.

Pour donner un ordre d'idée, dans le cas de l'exemple 3 avec la quantité d'énergie maximale évoquée ici, plus de **51 000 000** comparaisons élémentaires (comparaisons des symboles à la racine des termes) sont effectuées.

5.2.5 Conclusion

Ces exemples illustrent le fait que l'algorithme cherche plusieurs solutions en prenant en compte les propriétés du domaine et des paramètres. Toutes les solutions n'ont pas la même qualité et un choix doit être fait parmi elles.

Ces exemples démontrent également, que toute personne, même non spécialiste de l'algèbre linéaire, peut ainsi obtenir une aide précieuse pour trouver les services qui vont résoudre son problème. Il n'a plus besoin de connaître le nom exact du service (nom qui

n'est pas toujours très explicite), ni ce à quoi font référence les paramètres. Il lui suffit de formuler sa requête pour que la solution soit trouvée.

Néanmoins, il reste des paramètres qui ne sont pas déterminés. Comme il a été vu précédemment, ils devront être traités « à la main » par l'utilisateur. Il est aussi envisageable, de coupler notre algorithme de courtage avec un autre outil qui permettrait de déterminer ces paramètres. Dans le cas de nos exemples, ces paramètres font principalement référence à des tailles de matrices.

5.3 L'optimisation

Le second domaine qui nous a intéressé est l'optimisation. Comme l'algèbre linéaire, il n'est pas utilisé que par des spécialistes du domaine. Le bénéfice de l'approche est moindre car les services décrits et requis sont des services élémentaires de minimisation et maximisation. De plus, les compositions de service sont plus rares et les équations ont ici un rôle limité dans la mesure où, dans notre représentation, elles ne sont appliquées que sur les contraintes.

Néanmoins, notre algorithme permet toujours de renvoyer plusieurs solutions en tenant compte des spécificités du domaine. Il permet également de connaître les services qui résolvent notre problème sans avoir à connaître les noms des fonctions.

5.3.1 Formalisation du domaine

Les problèmes d'optimisation sont des problèmes de minimisation et maximisation de fonctions. Les types et opérateurs sont donc ceux décrivant les fonctions et contraintes.

Comme dans le cas de l'algèbre linéaire, nous nous positionnons dans l'ensembe des réels $\mathbb{R}$, mais la description s'étend sans difficulté au cas complexe.

5.3.1.1 Les types

Les principaux types sont ceux décrivant les fonctions (*funVector->Vector*, ...), les contraintes (*constraint*) et les éléments manipulés par les fonctions et les contraintes (*Real*, *Matrix*, ...).

L'ensemble des types est développé dans l'annexe C.1 (page 167).

5.3.1.2 Les opérateurs

Les opérateurs sont les opérateurs de minimisation et maximisation (*min*, *max*, ...), les opérateurs permettant de décrire les fonctions (->), ceux qui décrivent les contraintes (<=, &, ...) et ceux qui agissent sur les éléments manipulés (*, +, ...).

L'ensemble des opérateurs et leurs signatures est disponible dans l'annexe C.2 (page 167).

5.3.1.3 Les équations

Les équations (annexe C.3, page 169) sont celles qui permettent de manipuler les contraintes. Elles comportent l'associativité de l'opérateur d'addition des contraintes (&) et la manipulation des contraintes vides.

5.3.2 Les bibliothèques

Les principaux services que les bibliothèques d'optimisation permettent de résoudre sont :

- la résolution de problèmes non linéaires, avec ou sans contraintes :
  $\min_x f(x)$
- la résolution de problèmes linéaires, avec contraintes :
  $\min_x c^T x$
- la résolution des problèmes aux moindres carrés, avec ou sans contraintes :
  $\min_x \frac{1}{2}\|Cx - d\|_2^2$
- la résolution de problèmes quadratiques, avec ou sans contraintes :
  $\min_x \frac{1}{2}x^T Hx + f^T x$

Dans le cas où il y a des contraintes, celles-ci sont de type :

- $Ax \leq b$
- $Ax = b$
- $l \leq x \leq u$

Nous nous sommes plus particulièrement intéressés à deux bibliothèques : la boîte à outils d'optimisation de Matlab [2] et le paquetage E04 de NAG [3].

5.3.2.1 La boîte à outils matlab

La boîte à outils Matlab est intéressante car très utilisée. Elle permet notamment :

- la résolution de problèmes non linéaires :
  - uni-variable avec contraintes ($fminbnd$)
  - multi-variables avec contraintes ($fmincon$)
  - multi-variables sans contrainte ($fminunc$)
- la résolution de problèmes linéaires avec contraintes ($linprog$)
- la résolution des problèmes aux moindres carrés avec contraintes ($lsqlin$ ou $lsqnonneg$)
- la résolution de problèmes quadratiques ($quadprog$)

La description de ces services dans notre formalisme se trouve en annexe C.4 (page 169). Remarquons que certains services apparaissent plusieurs fois avec des paramètres différents. Cela est lié au fait que certains paramètres sont optionnels.

5.3.2.2 E04 - NAG

Le paquetage E04, *Minimizing or Maximizing a Function*, du NAG, *Numerical Algorithms Group* est la seconde bibliothèque qui nous a intéressée. C'est également une librairie très utilisée. Elle illustre, tout comme le BLAS ou LAPACK, le fait que le nom de la procédure n'est pas toujours intuitif et ne peut pas servir de base à une recherche. Ces noms répondent à une norme spécifique de la librairie à laquelle ils appartiennent. Mais chaque librairie ayant la sienne, d'éventuels critères généraux ne peuvent pas être dégagés simplement pour être utilisés dans un cadre générique.

Ce paquetage permet notamment :

- la résolution de problèmes non linéaires :
  - uni-variable avec contraintes ($E04ABF$ ou $E04BBF$)
  - multi-variables avec contraintes ($E04JYF$)

[2] `http://www.mathworks.com/access/helpdesk/help/toolbox/optim/optim.shtml`
[3] `http://www.csc.fi/cschelp/sovellukset/math/nag/NAGdoc/fl/html/E04_fl19.html`

  - multi-variables sans contrainte ($E04DGF$)
- la résolution de problèmes linéaires avec contraintes ($E04MFF$)
- la résolution des problèmes aux moindres carrés avec contraintes ($E04NCF$)
- la résolution de problèmes quadratiques ($E04NFF$)

La description de ces services dans notre formalisme se trouve en annexe C.5 (page 170). Ces services sont en fait plus complexes et résolvent plusieurs types de problèmes. Pour fixer le type de problème que nous souhaitons traiter, il faut préalablement faire appel à une autre procédure, en fixant les paramètres décrivant le problème à résoudre. Nous avons choisi de décrire le cas par défaut.

5.3.3 Exemples

Nous allons maintenant donner quelques exemples d'utilisation. Les recherches sont effectuées dans les deux bibliothèques que nous avons décrites.

5.3.3.1 Exemple 1

L'utilisateur veut minimiser une fonction f uni-variable sur l'intervalle $[a, b]$.
L'algorithme va alors trouver comme solution avec le paquetage E04 :

```
p1=?;
E04ABF(f, ?, ?, a, b, ?, ?, p1, ?);
p1;
```

Si f est dérivable, l'algorithme va trouver comme solutions :

```
p1=?;
E04ABF(f, ?, ?, a, b, ?, ?, p1, ?);
p1;
```

qui est celle du cas général et :

```
p1=?;
E04BBF(f, ?, ?, a, b, ?, ?, p1, ?, ? );
p1;
```

qui utilise un algorithme diffèrent basé sur la dérivée de f.
Dans ces deux cas, l'algorithme doit être paramétré avec :
- nombre d'équations autorisées : 0
- taille maximale des transformations : 0
- profondeur de composition : 0

Dans les deux cas, les solutions sont obtenues **instantanément** et l'algorithme ne trouve pas d'autre solution.

5.3.3.2 Exemple 2

L'utilisateur veut minimiser une fonction f multi-variables sur « l'intervalle » $[a, b]$.
L'algorithme va alors trouver comme solutions :

```
p1 = fmincon(f, ?, [ ], [], [ ], [], a, b );
p1;
```

et

```
p1=?;
E04JYF(?, ?, f, a, b, ?, p1, ?, ?, ?, ?, ?, ?, ? );
p1;
```

L'algorithme doit être paramétré avec :
- nombre d'équations autorisées : 4
- taille maximale des transformations : 1
- profondeur de composition : 0

La solution est obtenue **instantanément**. L'algorithme ne trouve pas d'autres solutions.

5.3.4 Conclusion

L'utilisation de notre mécanisme en optimisation montre d'abord que le choix de décrire les domaines à l'aide d'un type abstrait algébrique, qui est naturel pour l'algèbre linéaire, est également adapté à d'autres domaines.

Même si les équations sont, sur nos exemples, utilisées dans un unique sens, elles le sont dans le sens le moins naturel : celui qui « complique » le terme. Appliquées dans ce sens, elles ne permettraient pas de construire un système de réécriture convergent et une forme normale ne pourrait pas être utilisée. Donc, notre approche du traitement des équations se justifie dans ce contexte également.

5.4 Interaction entre l'algèbre linéaire et l'optimisation

Nous avons vu précédemment, section 4.11 (page 114), que l'algorithme a été étendu pour pouvoir travailler dans plusieurs domaines. Nous allons illustrer cette fonctionnalité dans le cas de l'algèbre linéaire et de l'optimisation.

5.4.1 L'intersection des domaines

Nous avons pu remarquer dans les descriptions que nous avons faites de l'algèbre linéaire et de l'optimisation que certains éléments d'un domaine se retrouvaient dans l'autre (les matrices par exemple). Pour que l'interaction entre des domaines soit possible, il faut que les descriptions données pour ces éléments soient identiques. Dans notre cas, nous retrouvons en commun les matrices, les matrices symétriques, certaines constantes et certains opérateurs sur les matrices.

Dans la formalisation de l'optimisation, nous ne trouvons pas une description aussi précise des matrices et de leurs opérateurs que dans le cas de l'algèbre linéaire. Cependant, si l'utilisateur à un problème d'optimisation à résoudre, un problème quadratique par exemple et que ce problème nécessite un travail en amont sur une matrice, il serait dommage qu'il ne puisse pas résoudre son problème. Il est alors intéressant de pouvoir basculer dans le domaine de l'algèbre linéaire pour traiter la matrice.

5.4.2 Un exemple : Machines à Vecteurs Supports

La SVM (Support Vector Machines) est une méthode de classification. Cette méthode est utilisée dans différents domaines et donc pas seulement par des utilisateurs experts des

bibliothèques d'algèbre linéaire et d'optimisation.

Au coeur de la SVM, la minimisation d'une forme quadratique sous contrainte d'égalités et d'inégalités linéaires doit être effectuée. Plus précisément le problème à résoudre est de la forme :

$$\min_x \frac{1}{2}x^T(Y * N * Y)x + f^T x,\ A_{eq}x \leq b_{eq},\ Ax = b$$

Le traitement de cette requête, en ne prenant en compte que l'algèbre linéaire ou que l'optimisation, n'aboutit pas. Mais en faisant interagir nos deux domaines nous sommes en mesure de résoudre ce problème.

En utilisant à la fois le BLAS, LAPACK, la boîte à outils d'optimisation de Matlab et le paquetage E04, l'algorithme de courtage renvoie parmi les solutions :

```
p1=O;
sgemm('n', 'n', ?, ?, ?, 1.0, Y, ?, N, ?, 1.0, p1, ? );
  //p1 <- Y*N
p2=O;
sgemm('n', 'n', ?, ?, ?, 1.0, p1, ?, Y, ?, 1.0, p2, ? );
  //p2 <- p1*Y
p3 = quadprog(p2, f, A, b, Aeq, beq );
  //résolution du problème quadratique
p3;
```

Pour trouver cette solution, l'algorithme doit être paramétré avec :

- nombre d'équations autorisées : 5
- taille maximale des transformations : 1
- profondeur de composition : 3

La solution est obtenue **instantanément**.

Dans cet exemple, lors du calcul, seules des équations de l'algèbre linéaire sont utilisées. Mais, si nous posons le problème :

$$\min_x \frac{1}{2}x^T(Y * N * Y)x,\ l \leq x \leq u$$

L'algorithme renvoie par exemple :

```
p1=O;
sgemm('n', 'n', ?, ?, ?, 1.0, Y, ?, N, ?, 1.0, p1, ? );
  //p1 <- Y*N
p2=O;
sgemm('n', 'n', ?, ?, ?, 1.0, p1, ?, Y, ?, 1.0, p2, ? );
  //p2 <- p1*Y
p3 = quadprog(p2, Oc, [ ], [], [ ], [], l, u );
  //résolution du problème quadratique
p3;
```

Pour obtenir cette solution, lors du calcul, des équations des deux domaines sont nécessaires.

Pour trouver cette solution, l'algorithme doit être paramétré avec :

- nombre d'équations autorisées : 7
- taille maximale des transformations : 1

– profondeur de composition : 3

Elle est obtenue en **2s**.

5.5 Conclusion

Nous venons de voir dans ce chapitre l'utilisation de notre approche dans deux domaines différents : l'algèbre linéaire et l'optimisation. Ces deux domaines ont été choisis car ce sont des domaines avec lesquels nous sommes familiers. De plus, nous travaillons au quotidien avec des spécialistes de ces domaines, ce qui a facilité notre travail.

Nous avons commencé à étudier d'autres domaines qui semblent bien se prêter à cette représentation : traitement du signal ou de l'image, statistiques et probabilité, ... Ces travaux sont encore à un stade embryonnaire.

Certes le temps de réponse n'est pas toujours négligeable, mais les réponses apportées par l'algorithme sont très pertinentes par rapport à la question posée. Ceci n'est pas toujours vrai avec d'autres types de descriptions.

L'interaction entre les domaines est aussi un point important. Il permet de travailler dans plusieurs domaines en parallèle, sans pour autant avoir à les fusionner en un seul. Un tel mécanisme permet de manipuler des domaines qui ne sont pas bien cloisonnés.

Nous allons maintenant conclure et présenter les différentes perspectives qui s'offrent à nous pour prolonger ces travaux.

Chapitre 6

Conclusion et Perspectives

6.1 Bilan

Nous avons proposé et mis en œuvre une approche pour le courtage de service développée pour pallier le manque de précision de certaines approches existantes. L'ajout de précision dans la description des services entraînent une plus grande complexité de la comparaison des descriptions, nous nous plaçons dans des domaines d'application particuliers afin de garder un temps de calcul raisonnable. La généricité de l'approche et la possibilité de faire interagir plusieurs domaines faiblement couplés assurent que ce positionnement dans un domaine particulier n'est pas une restriction trop contraignante.

Nos travaux s'inspirent du domaine des spécifications formelles en génie logiciel. Plus précisément, la description choisie est basée sur les spécifications algébriques qui offrent un bon compromis entre la facilité d'expression, la richesse sémantique et la construction d'algorithmes de comparaison avec des propriétés correspondant à nos besoins. De plus, cette notation est particulièrement adaptée à la description des services de calcul scientifique qui constituent le domaine d'applications principal de nos travaux. Elle est composée d'une signature hétérogène, avec sous-typage et surcharge, et d'un ensemble d'équations. Cet ensemble d'équations permet de définir les propriétés des opérateurs. Les équations offrent une richesse de description, mais elles rendent la comparaison plus complexe. En effet, la comparaison de deux termes modulo une théorie équationnelle (filtrage équationnel ou unification équationnelle) est indécidable. Il a donc fallu mettre en place d'une part des techniques de comparaison semi-décidable et d'autre part des stratégies particulières pour que nos algorithmes terminent. Plus précisément, notre objectif est d'arriver à construire chaque réponse positive en un temps fini, et d'autre part que l'algorithme doit générer toutes les solutions finies avant d'essayer de générer une solution infinie. Notre approche consiste à associer une consommation d'énergie à la construction d'une solution. La quantité d'énergie correspondant au nombre d'équations qui peuvent être appliquées et au nombre de services qui peuvent être composés.

Nous avons défini formellement à partir de la solution dérivée des spécifications algébriques, l'ensemble des services qui peuvent être rendus dans un domaine. Nous avons proposé deux algorithmes avec des caractéristiques différentes. Le premier s'inspire des travaux de Gallier et Snyder sur l'unification équationnelle et traite les sous-problèmes de manière indépendante en synthétisant les résultats pour conclure à l'existence d'une solution globale ensuite. Le second repose sur une définition constructive des solutions recherchées

qui répond au fait que la technique de Gallier et Snyder construit un sur-ensemble de ces solutions et qu'il était complexe de restreindre leur approche à nos objectifs tout en préservant la propriété de complétude de leur système. Cet algorithme traite les sous-problèmes en séquence et construit la solution au fur et à mesure. Cela permet d'obtenir des performances plus satisfaisantes. Ces deux algorithmes génèrent les solutions finies dont le coût est inférieur à une quantité d'énergie donnée, quantifiée en terme de profondeur de composition de services et en coût d'application d'équations.

Les algorithmes s'appuyant sur des approches formelles, nous avons étudié leur correction et leur complétude. Les preuves des corrections des deux algorithmes sont réalisées. Elles permettent de valider formellement l'approche. La complétude du second a également été prouvée pour certaines catégories d'équations. Quelques optimisations sont aussi proposées (simplification de la requête, pas de relance sur la requête, ...) pour permettre de trouver d'abord les services les plus intéressants ou de ne pas en calculer certains que nous savons moins intéressants. C'est l'utilisateur qui choisit, ou non, d'utiliser ces optimisations.

Des exemples concrets en algèbre linéaire et en optimisation ont été développés afin de montrer les bénéfices d'une telle approche. Ces exemples illustrent bien les capacités de l'algorithme à composer les méthodes et à renvoyer des solutions pertinentes, répondant effectivement au problème.

6.2 Perspectives

Nos travaux futurs se présentent sous sept angles à court et moyen terme :
- à court terme et en restant dans le même contexte, nous souhaitons :
  - finaliser les preuves de complétude du second algorithme ;
  - améliorer les performances de l'algorithme ;
  - intégrer notre algorithme à l'intergiciels de grille DIET ;
  - intégrer notre algorithme à TLSE, au niveau de PRUNE et WEAVER ;
  - envisager son intégration au projet LEGO ;
- à court terme et en changeant de contexte, nous envisageons :
  - étendre notre mécanisme à la manipulation de documents XML ;
- à moyen terme et en restant dans le même contexte, nous voulons :
  - compléter la description pour couvrir un spectre plus large des propriétés des services.

6.2.1 Finalisation des preuves de complétude du second algorithme

Nous avons prouvé que le second algorithme est complet pour certaines formes d'équations, et pour d'autres formes dans un sens particulier d'application. La complétude n'est certes pas la propriété la plus importante pour une utilisation pratique de nos travaux et tous les exemples traités ont exhibé toutes les solutions attendues. Néanmoins, nous souhaitons poursuivre l'étude de cette propriété pour dégager un plus grand nombre de catégories d'équations pour lesquelles la complétude est assurée. Nous pourrons ainsi informer le spécialiste qui défini un domaine des risques d'incomplétude liée à certaines équations. De toutes façons, nous avons montré que notre algorithme génère au moins toutes les solutions qui n'utilisent pas les équations suspectes.

Cette étude nous amènera peut être à faire des modifications dans la formalisation et dans l'algorithme.

6.2.2 Améliorations des performances de l'algorithme

Un mode plus incrémental doit être développé afin que les solutions soient proposées à l'utilisateur au fur et à mesure de leur découverte. À terme, il faudrait que la quantité d'énergie disparaisse des paramètres de l'algorithme. L'utilisateur décidera simplement d'arrêter les recherches quand il sera satisfait ou qu'il n'aura plus le temps d'attendre de meilleures solutions. Le problème est de trouver une façon efficace de reprendre les recherches dans une branche de l'arbre de recherche sans refaire des calculs déjà effectués. La difficulté majeure est le coût du stockage des étapes intermédiaires. Nous pouvons aussi envisager d'enrichir la quantité d'énergie en ajoutant un paramètre correspondant au nombre maximum de solutions que l'utilisateur souhaite obtenir. Une fois cette valeur atteinte, le calcul sera stoppé.

Il faudrait également améliorer les performances de l'algorithme. Pour cela, il faudrait diminuer le nombre de branches parcourues par l'algorithme et réussir à trouver des mécanismes pour détecter les branches menant à des solutions non pertinentes ou déjà trouvées. Il est également envisageable, quand un mode incrémental efficace aura été mis en place, de changer l'ordre de recherche. Actuellement, aucune priorité n'est appliquée sur les services ou les équations. Il est possible de le faire pour obtenir des solutions dans un ordre différent. Ceci sera particulièrement intéressant en bornant également le nombre de solutions recherchées.

La plupart des branches menant à l'échec, il est intéressant de détecter au plus vite ces échecs afin de ne pas poursuivre le calcul dans cette direction. Que ce soit au niveau du traitement des frères, dans le cas d'une décomposition, ou au niveau du choix de l'ordre dans lequel sont traitées les relances (pour les compositions de services), une détection précoce des erreurs permettrait de gagner un temps de calcul très important. Nous pouvons par exemple commencer par comparer les problèmes les plus complexes (avec les termes les plus profonds), car ceux-ci ont le plus de chance d'échouer. Néanmoins, leur traitement est complexe et demande beaucoup de temps. Par contre, les problèmes simples ont moins de chance d'échouer mais s'ils échouent, cet échec surviendra plus rapidement. Il faudra donc trouver un compromis entre les chances d'échec et les temps de calcul pour obtenir ces échecs. La seule certitude est que les échecs sont provoqués soit par un manque d'énergie, soit par des problèmes comparant deux constantes différentes. Dans les cas que nous évoquons, les problèmes possèdent tous la même quantité d'énergie disponible, elle ne pourra donc pas intervenir dans la fonction de poids. Par contre, nous aurons intérêt à comparer en premier les termes principalement composés de constantes.

Pour améliorer les performances, une suppression du travail redondant doit donc être effectuée grâce à la mise en place d'un mécanisme de cache efficace. Il faudra également trouver un compromis entre le temps de calcul, le temps de recherche dans le cache et l'espace mémoire nécessaire au cache.

6.2.3 Intégration dans des intergiciels

L'objectif premier était de réaliser un algorithme de courtage de service sur la grille. Il faut donc intégrer cet algorithme à un intergiciel de grille. Dans le cadre du projet TLSE, l'intergiciel DIET [CDL+02] est utilisé. Un travail de réflexion sur cette intégration a déjà été mené et exposé à la communauté du parallélisme [HPD04]. L'interaction avec un intergiciel permettra de tirer profit du mécanisme d'ordonnancement de l'intergiciel afin de trier les différentes solutions renvoyées par l'algorithme. L'intergiciel tiendra en effet compte des propriétés du réseau, de la charge des machines, ...

6.2.3.1 DIET

Comme il a été dit précédemment, ce travail est lié au projet TLSE qui s'appuie sur l'intergiciel DIET. Ce dernier s'appuie lui-même sur le standard GridRPC [SLD+04] pour l'interface client. Ce paradigme est proche du modèle RPC (*Remote Procedure Call* ou appel de procédure à distance). Les environnements qui les réalisent sont appelés des serveurs de calculs ou NES (*Network Enabled Servers*). Plusieurs outils offrant cette fonctionnalité comme NetSolve [1], NINF [2], DIET [3], NEOS [4], ou RCS [AGM97] sont déjà disponibles.

Dans [MNSS00], les auteurs donnent un état de l'art des environnements basés sur des NES. Ils sont composés de cinq types de composants différents : les **clients** qui soumettent les problèmes aux serveurs, les **serveurs** qui résolvent les problèmes soumis par les clients, des **moniteurs** qui récupèrent des informations sur l'état des ressources de calcul et les stockent dans une **base de données** qui contient aussi des informations concernant les ressources matérielles et logicielles, et enfin un **ordonnanceur** (appelé **agent** dans DIET) qui choisit un serveur approprié en fonction du problème soumis et des informations contenues dans la base de données.

DIET répartit le travail de l'agent selon une nouvelle organisation. L'agent est ainsi remplacé par un ensemble d'agents organisés selon deux approches : une approche hiérarchique favorisant l'efficacité de l'ordonnancement (Figure 6.1) et une approche multi-agents de type pair-à-pair améliorant la robustesse du système associé (Figure 6.2). Cette répartition du rôle de l'agent offre divers avantages :

- une meilleure répartition de la charge entre les différents agents ;
- une plus grande stabilité du système (si un des éléments venait à s'arrêter, les autres éléments pourraient se réorganiser pour le remplacer) ;
- une gestion simplifiée en cas de passage à l'échelle (l'administration de chaque groupe de serveurs et des agents associés peut être déléguée).

Un **client** est une application qui utilise DIET pour résoudre des problèmes.

Un **Master Agent (MA)** MA est directement relié aux clients. Il reçoit des requêtes de calcul des clients et choisit un (ou plusieurs) SeD qui sont capables de résoudre le problème en un temps raisonnable. Un MA possède les mêmes informations qu'un LA, mais il a une vue globale (et de haut niveau) de tous les problèmes qui peuvent être résolus et de toutes les données qui sont distribuées dans tous ses sous-arbres.

Un **Leader Agent (LA)** LA a pour but de diffuser les requêtes et les informations entre les MA et les SeD. Il tient à jour une liste des requêtes en cours de traitement et, pour

[1] `http://www.cs.utk.edu/netsolve/`
[2] `http://ninf.etl.go.jp/`
[3] `http://graal.ens-lyon.fr/DIET`
[4] `http://www-neos.mcs.anl.gov/`

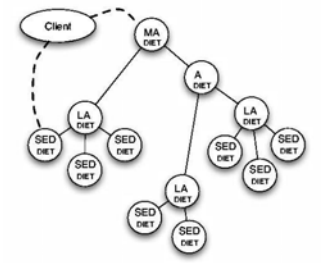


FIG. 6.1 – Architecture de DIET.

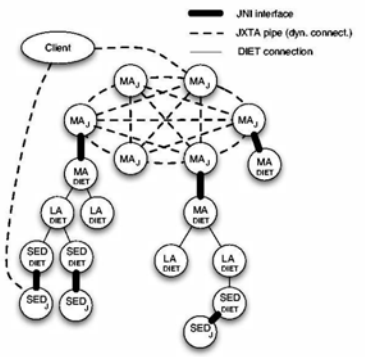


FIG. 6.2 – Architecture de DIET$_J$ avec connexions P2P.

chacun de ses sous-arbres, le nombre de serveurs pouvant résoudre un problème donné, ainsi que des informations à propos des données.

Un **Server Daemon (SeD)** est le point d'entrée d'un serveur de calcul. Il se trouve sous la responsabilité d'un LA. Il tient à jour une liste des données disponibles sur un serveur (éventuellement avec leur distribution et le moyen d'y accéder), une liste des problèmes qui peuvent y être résolus, et toutes les informations concernant sa charge. Sur une machine parallèle, un SeD sera donc installé sur le frontal de cette machine.

DIET fonctionne selon le mode opératoire suivant. Un nouveau client doit d'abord contacter un Master Agent (le plus approprié : en distance réseau par exemple) et lui soumettre un problème. Le Master Agent propage une requête dans ses sous-arbres afin de trouver à la fois les données impliquées (parfois issues de calculs précédents et donc déjà présentes sur certains serveurs lorsque la persistance est activée) et les serveurs capables de résoudre l'opération demandée. Lorsque la réponse revient au Master Agent, il renvoie l'adresse du serveur choisi au client (il est également possible de renvoyer une liste bornée des meilleurs serveurs au client).

6.2.3.2 Intégration dans DIET

En plaçant l'algorithme de courtage au niveau des Master Agents (qui sont en relation avec le client), nous aurons les informations nécessaires pour effectuer les comparaisons.

L'architecture de DIET permettra donc d'échanger avec l'environnement de courtage, des informations importantes. Parmi celles-ci se trouveront la disponibilité des services, l'encombrement des machines, ... En retour, DIET recevra les informations sur la possibilité d'utiliser tel ou tel service ou composition de services pour résoudre un problème soumis par un client. Nous construirons ainsi plusieurs plans d'exécutions, DIET évaluera ensuite les coûts en terme de temps d'exécution et de communication. Si DIET n'est pas satisfait par les solutions obtenues (par exemple, les solutions nécessitent des temps d'exécution trop importants), il pourra relancer l'algorithme de courtage pour obtenir d'autres solutions.

6.2.3.3 MATLAB et SciLAB

Un de nos objectifs futurs est également d'intégrer un mécanisme de courtage similaire à des environnements de calcul scientifique tels MATLAB[5] ou SciLAB[6] pour permettre aux utilisateurs de tirer profit des services disponibles dans les bibliothèques ou à travers la grille, quand cela est adéquat et de façon transparente. Ce sujet a donné lieu à une proposition acceptée de projet en commun avec l'université de Iervan en Arménie.

En effet, depuis 2000 et la version 5.3, Matlab utilise LAPACK et le BLAS optimisé pour les calculs matriciels. Les commandes dans Matlab n'ont pas changées mais les temps d'exécutions et les résultats numériques sont améliorés.

À partir d'une commande simple de Matlab ($x = A\backslash b$ par exemple), une analyse des données est effectuée pour déterminer les propriétés des éléments et déterminer la méthode la plus adéquate à appeler. Ceci est effectué de manière ad-hoc pour un jeu de services figé avec la version des outils.

Dans le contexte de grille et avec plus de librairies disponibles qu'uniquement LAPACK, nous pourrions utiliser notre mécanisme de courtage au sein de Matlab, soit de façon transparente pour l'utilisateur caché derrière les opérateurs standard, soit explicitement. Cela permettrait de choisir au mieux l'algorithme à utiliser et de pouvoir améliorer encore les performances et les précisions numériques. Cela permettrait, par exemple, après extension de notre description de l'algèbre linéaire, de faire appel à des bibliothèques spécialisées pour les matrices creuses, ce que LAPACK ne traite pas. Cette extension pourrait également profiter de la combinaison avec TLSE qui offre une description à base de mots-clés structurés.

6.2.4 Intégration au projet TLSE

Comme il a été dit précédemment (section 5.2.1.1, page 123), le but du projet TLSE[7] est de développer un site d'expertise pour la résolution de systèmes linéaires creux par méthodes directes. Il s'agit donc de résoudre $Ax = b$, où A est une matrice creuse, en utilisant des algorithmes directs de résolution. Plusieurs algorithmes peuvent être utilisés pour résoudre le même système linéaire. Ils utilisent les mêmes paramètres fonctionnels en entrée : A et b et produisent tous la même sortie fonctionnelle : x. Cependant, ils n'ont pas toujours le même ensemble de paramètres en entrée et sortie pour le contrôle de l'algorithme. Ils proposent également des métriques d'exécution (temps d'exécution, mémoire utilisée, nombre de flops, ...) qui ne sont pas forcément identiques. Ces paramètres non-fonctionnels sont ceux que notre algorithme ne sait pas exploiter, puisque nous nous sommes intéressés uniquement à l'aspect fonctionnel sans rentrer dans les détails des algorithmes (contrairement à certains projets évoqués en introduction, comme FLAME ou Falcon).

Dans le projet TLSE, ces paramètres sont décrits à l'aide de méta-données, appelées *paramètres abstraits*, qui permettent d'exprimer toutes les implantations possibles d'un service. Il existe deux sortes de services dans le projet TLSE :

- les services de calculs (qui correspondent aux services que nous manipulons) ;
- les scénarios, qui décrivent le processus d'expertise.

Les paramètres abstraits sont utilisés pour exprimer les contraintes et / ou relations exploitées par les scénarios d'expertise au sein de WEAVER pour choisir des outils pour

[5] `http://www.mathworks.com/`

[6] `http://www.scilab.org/`

[7] `http://www.enseeiht.fr/lima/tlse/index.html`

effectuer les expériences demandées par le scénario dans le but de conseiller le client dans le choix d'un outil. Il faut noter que TLSE offre un mécanisme de description des méta-données qui permet de définir l'ensemble des méta-données propres à un domaine applicatif à travers l'interface web, WEBSOLVE, puis d'exploiter ces méta-données pour décrire les services, les outils, les scénarios et les demandes d'expertise, également à travers WEBSOLVE. Il est aussi possible d'exprimer des dépendances entre les différentes métriques et paramètres de contrôle.

Dans le projet TLSE, les outils sont intégrés à l'aide de la description des paramètres abstraits et d'un adaptateur logiciel qui traduit les méta-données en paramètres et résultats effectifs des outils.

En ajoutant notre description fonctionnelle aux aspects non fonctionnels exprimés au sein du projet TLSE, nous pourrions déterminer les paramètres que nous ne savons pas déterminer pour l'instant et nous permettrons à TLSE d'exploiter une description mathématique rigoureuse des services.

6.2.5 Participation au projet LEGO

LEGO (League for Efficient Grid Operation) est un projet financé par l'ACI Calcul Scientifique et Grille. Son objectif principal est la combinaison des résultats de plusieurs projets autour du développement d'applications parallèles réparties sur la grille et de valider les résultats de cette combinaison sur plusieurs applications réelles dont TLSE. Les projets considérés sont :

- DIET : intergiciel de type GRID-RPC ;
- JuxMem[8] (Juxtaposed Memory) [ABJ05] : intergiciel de partage mémoire sur la grille ;
- GridCCM[9] [DPPR04] : adaptation de l'intergiciel CCM (Corba Component Model) pour permettre à des composants d'applications à base de passage de messages (PVM, MPI) de partager des informations efficacement sans passer systématiquement par des appels de méthode Corba coûteux ;
- Madeleine[10] [ABMN02] : bibliothèque efficace de manipulation de processus et d'entrées/sorties.

Nous intervenons à deux titres dans le projet, d'une part en temps que fournisseur d'un cas d'application (TLSE) et d'autre part au niveau intergiciel pour la prise en compte de composants patrimoniaux, c'est-à-dire de composants qui ne peuvent pas être modifiés. TLSE offre déjà plusieurs moyens pour intégrer ce type d'outils pré-existants à partir de description à base de mots-clés. Nos travaux permettront d'une part d'y ajouter des informations sur la fonction mathématique calculée, et d'autre part de permettre un courtage de composants basé sur cette description, soit lors du développement de l'application, soit dynamiquement lors de son exécution en collaboration avec DIET.

6.2.6 Manipulation de documents XML

Les DTD peuvent être vues comme des spécifications algébriques sans équation et un document XML respectant une DTD comme un terme sur l'algèbre associée. Seules les répétitions d'éléments (opérateur de Kleene) ne sont pas traitées dans les spécifications

[8] `http://juxmem.gforge.inria.fr/`

[9] `http://www.irisa.fr/paris/Gridccm/`

[10] `http://runtime.futurs.inria.fr/mpi/`

algébriques, mais peuvent être ajoutées à l'aide d'un opérateur de séquencement qui forme un monoïde libre.

Actuellement, les comparaisons de documents XML se font sur l'égalité stricte des documents. Pourtant, deux documents respectant la même DTD et possédant les mêmes informations peuvent avoir une structure différente si la DTD en a laissé la possibilité. En effet, les DTD sont parfois moins restrictives que ce qu'elles devraient. Si les documents qui sont comparés sont engendrés par le même outil ou la même personne, ils auront probablement la même structure et la comparaison sera alors simple. Mais si nous souhaitons comparer deux documents issus de deux sources différentes, alors la structure des documents pourra être différente.

Si des informations sont ajoutées à la DTD pour indiquer une égalité entre deux documents (ou deux parties d'un document), les comparaisons pourront se faire modulo ces égalités.

Si ces informations sont représentées sous forme d'équations, notre mécanisme de courtage peut alors être appliqué. C'est un cas un peu particulier car nous ne souhaitons pas trouver toutes les correspondances, mais seulement savoir si les documents sont égaux. En fait, nous aurons des constantes dans les deux membres et nous souhaitons juste savoir si l'algorithme de courtage échoue (auquel cas les documents sont différents) ou s'il réussit (auquel cas les documents sont identiques).

Nous pouvons également utiliser notre algorithme pour faire du filtrage de documents (à la XQuery[11]) modulo une théorie équationnelle quelconque définie sur la DTD. Dans ce cas, les différentes solutions peuvent se révéler intéressantes.

6.2.7 Enrichissement de la description des services

Actuellement, les propriétés sur les paramètres sont représentées à l'aide de types. Il peut être envisagé de remplacer ou de compléter cette représentation par une autre à l'aide de méta-données plus élaborées. En effet, ne pouvant pas définir le fait qu'un paramètre a plusieurs propriétés en utilisant plusieurs types, nous avons défini un type pour chaque combinaison de propriétés. Cela est assez lourd et pourrait être évité en définissant des propriétés sur les paramètres.

Nous avons vu dans le cas de l'algèbre linéaire que nous nous sommes restreints au cas des nombres réels. Pourtant tout ce que nous avons décrit dans le cas réel existe également dans le cas complexe, mais aussi en simple ou double précision. La généricité serait alors très intéressante car elle éviterait de dupliquer toute la description. Il y aurait des matrices avec des éléments de type $'a$ et les services agiraient sur des éléments de type $Real\ Matrix$ ou $Complex\ Matrix$. Une intégration immédiate de cette généricité serait la duplication automatique (c'est l'algorithme qui le fait, pas l'utilisateur) des types, des opérateurs et des équations en instanciant $'a$ par toutes les valeurs qu'il peut prendre. Mais il existe sûrement une méthode plus efficace pour traiter cette généricité, en la gardant explicitement et non en la faisant disparaître.

Il reste certains paramètres qui ne peuvent pas être déterminés, car n'intervenant pas dans la description fonctionnelle du service. Plusieurs solutions sont envisageables pour résoudre ce problème. L'algorithme actuel pourrait être couplé avec un autre algorithme basé

[11] `http://www.w3.org/TR/2006/CR-xquery-20060608/`

sur une autre description et qui pourrait exprimer ces paramètres. Dans le cas de l'utilisation de l'algorithme par un utilisateur « humain », il peut être envisagé de lui laisser à charge le choix de ces paramètres. Pour lui éviter une recherche trop contraignante, il faudrait lui fournir la documentation des services. Il peut également être envisagé de définir des valeurs par défaut pour ces paramètres.

Pour traiter ces paramètres, nous pouvons également utiliser une description plus complète, qui nous permettrait par exemple de représenter les tailles des matrices. Nous avons vu que cela était possible avec la logique de Hoare. Au lieu d'avoir, comme actuellement, en pré-conditions des contraintes de typage et en post-conditions des contraintes de typage et la fonctionnalité rendue, nous aurions un ensemble plus riche de pré- et post-conditions. Certains éléments de description pourraient toujours être décrits à l'aide de spécifications algébriques. D'autres pourraient être introduits avec la contrainte de disposer d'algorithmes semi-décidables. Le mécanisme de comparaison sera conservé sur la fonctionnalité mais devra être étendu pour traiter les autres informations.

Nous avons vu que, dans RDF, la description associée à un champ peut être quelconque. Nous pourrions donc intégrer notre description à RDF. Cela permettrait un courtage plus précis.

Annexe A

Fichiers XML

A.1 Domaine

```
<?xml version="1.0" encoding="UTF-8"?>
<domain>
  <domainName>DomaineExemple</domainName>
  <sortes>
    <defsorte>
      <name>Matrix</name>
    </defsorte>
  </sortes>
  <operators>
    <operator>
      <operatorName>+</operatorName>
      <parameters>
        <parameter>
          <sorte>Matrix</sorte>
        </parameter>
        <parameter>
          <sorte>Matrix</sorte>
        </parameter>
      </parameters>
      <notation>infix</notation>
      <commutativity>true</commutativity>
      <return>
        <sorte>Matrix</sorte>
      </return>
    </operator>
    <operator>
      <operatorName>*</operatorName>
      <parameters>
        <parameter>
          <sorte>Matrix</sorte>
        </parameter>
        <parameter>
          <sorte>Matrix</sorte>
```

```
        </parameter>
      </parameters>
      <notation>infix</notation>
      <commutativity>false</commutativity>
      <return>
        <sorte>Matrix</sorte>
      </return>
    </operator>
    <operator>
      <operatorName>0</operatorName>
      <parameters />
      <notation>prefix</notation>
      <commutativity>false</commutativity>
      <return>
        <sorte>Matrix</sorte>
      </return>
    </operator>
    <operator>
      <operatorName>I</operatorName>
      <parameters />
      <notation>prefix</notation>
      <commutativity>false</commutativity>
      <return>
        <sorte>Matrix</sorte>
      </return>
    </operator>
  </operators>
  <equalities>
    <equality>
      <rw>true</rw>
      <term>
        <expression>
          <operatorName>*</operatorName>
          <terms>
            <term>
              <variable>
                <name>x</name>
                <sorte>Matrix</sorte>
              </variable>
            </term>
            <term>
              <constant>
                <value>I</value>
                <sorte>Matrix</sorte>
              </constant>
            </term>
          </terms>
        </expression>
```

```
      </term>
      <term>
        <variable>
          <name>x</name>
          <sorte>Matrix</sorte>
        </variable>
      </term>
    </equality>
    <equality>
      <rw>false</rw>
      <term>
        <expression>
          <operatorName>+</operatorName>
          <terms>
            <term>
              <variable>
                <name>x</name>
                <sorte>Matrix</sorte>
              </variable>
            </term>
            <term>
              <expression>
                <operatorName>+</operatorName>
                <terms>
                  <term>
                    <variable>
                      <name>y</name>
                      <sorte>Matrix</sorte>
                    </variable>
                  </term>
                  <term>
                    <variable>
                      <name>z</name>
                      <sorte>Matrix</sorte>
                    </variable>
                  </term>
                </terms>
              </expression>
            </term>
          </terms>
        </expression>
      </term>
      <term>
        <expression>
          <operatorName>+</operatorName>
          <terms>
            <term>
              <expression>
```

```
                <operatorName>+</operatorName>
                <terms>
                  <term>
                    <variable>
                      <name>x</name>
                      <sorte>Matrix</sorte>
                    </variable>
                  </term>
                  <term>
                    <variable>
                      <name>y</name>
                      <sorte>Matrix</sorte>
                    </variable>
                  </term>
                </terms>
              </expression>
            </term>
            <term>
              <variable>
                <name>z</name>
                <sorte>Matrix</sorte>
              </variable>
            </term>
          </terms>
        </expression>
      </term>
    </equality>
  </equalities>
</domain>
```

A.2 Bibliothèque

```
<?xml version="1.0" encoding="UTF-8"?>
<library>
  <libraryName>LibrairieExemple</libraryName>
  <domainName>DomaineExemple</domainName>
  <services>
    <service>
      <declaration>
        <serviceName>addition3</serviceName>
        <variables>
          <variable>
            <name>x</name>
            <sorte>Matrix</sorte>
          </variable>
          <variable>
            <name>y</name>
            <sorte>Matrix</sorte>
```

```
        </variable>
        <variable>
          <name>z</name>
          <sorte>Matrix</sorte>
        </variable>
      </variables>
      <return>
        <sorte>Matrix</sorte>
      </return>
    </declaration>
    <specification>
      <pure>
        <term>
          <expression>
            <operatorName>+</operatorName>
            <terms>
              <term>
                <variable>
                  <name>x</name>
                  <sorte>Matrix</sorte>
                </variable>
              </term>
              <term>
                <expression>
                  <operatorName>+</operatorName>
                  <terms>
                    <term>
                      <variable>
                        <name>y</name>
                        <sorte>Matrix</sorte>
                      </variable>
                    </term>
                    <term>
                      <variable>
                        <name>z</name>
                        <sorte>Matrix</sorte>
                      </variable>
                    </term>
                  </terms>
                </expression>
              </term>
            </terms>
          </expression>
        </term>
      </pure>
    </specification>
    <cost>0</cost>
  </service>
```

```
<service>
  <declaration>
    <serviceName>multiplication</serviceName>
    <variables>
      <variable>
        <name>x</name>
        <sorte>Matrix</sorte>
      </variable>
      <variable>
        <name>y</name>
        <sorte>Matrix</sorte>
      </variable>
    </variables>
    <return>
      <sorte>Matrix</sorte>
    </return>
  </declaration>
  <specification>
    <pure>
      <term>
        <expression>
          <operatorName>*</operatorName>
          <terms>
            <term>
              <variable>
                <name>x</name>
                <sorte>Matrix</sorte>
              </variable>
            </term>
            <term>
              <variable>
                <name>y</name>
                <sorte>Matrix</sorte>
              </variable>
            </term>
          </terms>
        </expression>
      </term>
    </pure>
  </specification>
  <cost>0</cost>
</service>
<service>
  <declaration>
    <serviceName>addmult</serviceName>
    <variables>
      <variable>
        <name>x</name>
```

```
        <sorte>Matrix</sorte>
      </variable>
      <variable>
        <name>y</name>
        <sorte>Matrix</sorte>
      </variable>
      <variable>
        <name>z</name>
        <sorte>Matrix</sorte>
      </variable>
    </variables>
    <return>
      <sorte>Matrix</sorte>
    </return>
  </declaration>
  <specification>
    <pure>
      <term>
        <expression>
          <operatorName>*</operatorName>
          <terms>
            <term>
              <variable>
                <name>x</name>
                <sorte>Matrix</sorte>
              </variable>
            </term>
            <term>
              <expression>
                <operatorName>+</operatorName>
                <terms>
                  <term>
                    <variable>
                      <name>y</name>
                      <sorte>Matrix</sorte>
                    </variable>
                  </term>
                  <term>
                    <variable>
                      <name>z</name>
                      <sorte>Matrix</sorte>
                    </variable>
                  </term>
                </terms>
              </expression>
            </term>
          </terms>
        </expression>
```

```
          </term>
        </pure>
      </specification>
      <cost>0</cost>
    </service>
  </services>
</library>
```

A.3 Requête

```
<?xml version="1.0" encoding="UTF-8"?>
<requete>
  <domainName>DomaineExemple</domainName>
  <term>
    <expression>
      <operatorName>+</operatorName>
      <terms>
        <term>
          <expression>
            <operatorName>*</operatorName>
            <terms>
              <term>
                <variable>
                  <name>a</name>
                  <sorte>Matrix</sorte>
                </variable>
              </term>
              <term>
                <variable>
                  <name>b</name>
                  <sorte>Matrix</sorte>
                </variable>
              </term>
            </terms>
          </expression>
        </term>
        <term>
          <expression>
            <operatorName>*</operatorName>
            <terms>
              <term>
                <variable>
                  <name>c</name>
                  <sorte>Matrix</sorte>
                </variable>
              </term>
              <term>
                <variable>
```

```
                    <name>d</name>
                    <sorte>Matrix</sorte>
                  </variable>
                </term>
              </terms>
            </expression>
          </term>
        </terms>
      </expression>
    </term>
</requete>
```

Annexe B

Formalisation de l'algèbre linéaire

B.1 Les types

- les caractères
  - Char
- les scalaires
  - Int, NzInt
  - Real, NzReal
- les vecteurs
  - Vector
  - VectorL (vecteurs lignes)
- les matrices
  - Matrix
  - InvMatrix (matrices inversibles)
  - SymMatrix (matrices symétriques)
  - SymInvMatrix (matrices symétriques et inversibles)
  - TriMatrix (matrices triangulaires)
  - LowTriMatrix (matrices triangulaires inférieures)
  - TriInvMatrix (matrices triangulaires et inversibles)
  - UnitTriMatrix (matrices unitaires triangulaires)
  - UpTriMatrix (matrices triangulaires supérieures)
  - UnitLowTriMatrix (matrices unitaires triangulaires inférieures)
  - LowTriInvMatrix (matrices triangulaires inférieures et inversibles)
  - UnitTriInvMatrix (matrices unitaires, triangulaires et inversibles)
  - UpTriInvMatrix (matrices triangulaires supérieures et inversibles)
  - UnitUpTriMatrix (matrices unitaires triangulaires supérieures)
  - UnitLowTriInvMatrix (matrices unitaires triangulaires inférieures et inversibles)
  - UnitUpTriInvMatrix (matrices unitaires triangulaires supérieures et inversibles)
  - DiagMatrix (matrices diagonales)
  - DiagInvMatrix (matrices diagonales inversibles)
  - SymDefPosMatrix (matrices symétriques définies positives)

B.2 Les opérateurs

- Addition :

  - + : Matrix × Matrix→ Matrix [comm]
  - + : SymMatrix × SymMatrix→ SymMatrix [comm]
  - + : DiagMatrix × DiagMatrix→ DiagMatrix [comm]
  - + : LowTriMatrix × LowTriMatrix→ LowTriMatrix [comm]
  - + : UpTriMatrix × UpTriMatrix→ UpTriMatrix [comm]
  - + : Vector × Vector→ Vector [comm]
  - + : Int × Int→ Int [comm]
  - + : Real × Real→ Real [comm]
- Soustraction :
  - - : Matrix × Matrix→ Matrix
  - - : SymMatrix × SymMatrix→ SymMatrix
  - - : DiagMatrix × DiagMatrix→ DiagMatrix
  - - : LowTriMatrix × LowTriMatrix→ LowTriMatrix
  - - : UpTriMatrix × UpTriMatrix→ UpTriMatrix
  - - : Vector × Vector→ Vector
  - - : Int × Int→ Int
  - - : Real × Real→ Real
- Négation :
  - ˜ : Int→ Int
  - ˜ : NzInt→ NzInt
  - ˜ : Real→ Real
  - ˜ : NzReal→ NzReal
  - ˜ : Vector→ Vector
  - ˜ : VectorL→ VectorL
  - ˜ : Matrix→ Matrix
  - ˜ : InvMatrix→ InvMatrix
  - ˜ : SymMatrix→ SymMatrix
  - ˜ : SymInvMatrix → SymInvMatrix
  - ˜ : TriMatrix→ TriMatrix
  - ˜ : LowTriMatrix→ LowTriMatrix
  - ˜ : TriInvMatrix→ TriInvMatrix
  - ˜ : UpTriMatrix→UpTriMatrix
  - ˜ : LowTriInvMatrix→ LowTriInvMatrix
  - ˜ : UpTriInvMatrix → UpTriInvMatrix
  - ˜ : DiagMatrix→ DiagMatrix
  - ˜ : DiagInvMatrix → DiagInvMatrix
- Multiplication :
  - * : Matrix × Matrix→ Matrix
  - * : Real × Matrix→ Matrix
  - * : Int × Matrix→ Matrix
  - * : DiagMatrix × DiagMatrix→ DiagMatrix
  - * : Real × DiagMatrix→ DiagMatrix
  - * : Int × DiagMatrix→ DiagMatrix
  - * : InvMatrix × InvMatrix→ InvMatrix
  - * : NzReal × InvMatrix→ InvMatrix
  - * : NzInt × InvMatrix→ InvMatrix
  - * : DiagInvMatrix × DiagInvMatrix→ DiagInvMatrix
  - * : NzReal × DiagInvMatrix→ DiagInvMatrix

  - * : NzInt × DiagInvMatrix→ DiagInvMatrix
  - * : LowTriInvMatrix × LowTriInvMatrix→ LowTriInvMatrix
  - * : NzReal × SymDefPosMatrix→ SymDefPosMatrix
  - * : NzInt × SymDefPosMatrix→ SymDefPosMatrix
  - * : NzReal × SymInvMatrix→ SymInvMatrix
  - * : NzInt × SymInvMatrix→ SymInvMatrix
  - * : Real × SymMatrix→ SymMatrix
  - * : Int × SymMatrix→ SymMatrix
  - * : NzReal × TriInvMatrix→ TriInvMatrix
  - * : NzInt × TriInvMatrix→ TriInvMatrix
  - * : Real × TriMatrix→ TriMatrix
  - * : Int × TriMatrix→ TriMatrix
  - * : UpTriInvMatrix × UpTriInvMatrix→ UpTriInvMatrix
  - * : Int × Int→ Int
  - * : NzInt × NzInt→ NzInt
  - * : NzReal × NzReal→ NzReal
  - * : Real × Real→ Real
- Inverse :
  - ^-1 :InvMatrix → InvMatrix
  - ^-1 :TriInvMatrix → TriInvMatrix
  - ^-1 :LowTriInvMatrix → UpTriInvMatrix
  - ^-1 :UnitLowTriInvMatrix → UnitUpTriInvMatrix
  - ^-1 :UpTriInvMatrix → LowTriInvMatrix
  - ^-1 :SymInvMatrix → SymInvMatrix
  - ^-1 :DiagInvMatrix → DiagInvMatrix
  - ^-1 :SymDefPosMatrix → SymDefPosMatrix
- Transposée :
  - ^T :Matrix → Matrix
  - ^T :TriMatrix → TriMatrix
  - ^T :SymMatrix → SymMatrix
  - ^T :InvMatrix → InvMatrix
  - ^T :LowTriMatrix → UpTriMatrix
  - ^T :UpTriMatrix → LowTriMatrix
  - ^T :TriInvMatrix → TriInvMatrix
  - ^T :LowTriInvMatrix → UpTriInvMatrix
  - ^T :UpTriInvMatrix → LowTriInvMatrix
  - ^T :SymInvMatrix → SymInvMatrix
  - ^T :DiagMatrix → DiagMatrix
  - ^T :DiagInvMatrix → DiagInvMatrix
  - ^T :SymDefPosMatrix → SymDefPosMatrix
  - ^T :Vector → VectorL
- Factorisations :
  - choleskyU :SymDefPosMatrix → UpTriInvMatrix
  - choleskyL :SymDefPosMatrix → LowTriInvMatrix
  - l :InvMatrix → UnitLowTriInvMatrix
  - u :InvMatrix → UpTriInvMatrix
  - pivlu :InvMatrix → Vector
- Normes :

  - norm1 :Vector → Real
  - norm2 :Vector → Real
- les autres opérateurs (op : en fonction de la valeur du caractère renvoie la matrice, sa transposée ou son hermitienne ; perm : permutation d'une matrice ; up : accès à la partie supérieure d'une matrice ; low : accès à la partie inférieure d'une matrice)
  - op :Char × Matrix → Matrix
  - op :Char × TriMatrix → TriMatrix
  - op :Char × SymMatrix → SymMatrix
  - op :Char × InvMatrix → InvMatrix
  - op :Char × TriInvMatrix → TriInvMatrix
  - op :Char × SymInvMatrix → SymInvMatrix
  - op :Char × DiagMatrix → DiagMatrix
  - op :Char × DiagInvMatrix → DiagInvMatrix
  - op :Char × SymDefPosMatrix → SymDefPosMatrix
  - perm :Vector × Matrix → Matrix
  - perm :Vector × InvMatrix → InvMatrix
  - up :InvMatrix → UpTriInvMatrix
  - low :InvMatrix → LowTriInvMatrix

- -1 : → NzInt
- 1 : → NzInt
- 0 : → Int
- -1.0 : → NzReal
- 1.0 : → NzReal
- 0.0 : → Real
- -I : → DiagInvMatrix
- I : → DiagInvMatrix
- O : → Matrix
- 'n' : → Char
- 't' : → Char
- 'l' : → Char
- 'u' : → Char
- 'r' : → Char

B.3 Les équations

- Éléments neutres :
  - => {a :} (a - 0)=a
  - => {a :} (a + 0)=a
  - => {a :} (a - 0.0)=a
  - => {a :} (a + 0.0)=a
  - => {a :} (a - O)=a
  - => {a :} (a + O)=a
  - => {a :} (1 * a)=a
  - => {a :} (a * 1)=a
  - => {a :} (I * a)=a
  - => {a :} (a * I)=a
  - => {a :} (1.0 * a)=a

  - => {a :} (a * 1.0)=a
- Éléments absorbants :
  - => {a :} (0.0 * a)=0.0
  - => {a :} (a * 0.0)=0.0
  - => {a :} (a * 0)=0
  - => {a :} (0 * a)=0
  - => {a :} (0.0 * a)=O
  - => {a :} (0 * a)=O
  - => {a :} (O * a)=O
  - => {a :} (a * O)=O
- Distribution / factorisation :
  - == {a :, b :} ((a * b) ^-1)=((b ^-1) * (a ^-1))
  - => {a :} ((a ^T) ^-1)=((a ^-1) ^T)
  - => {a :} ((˜ a) ^T) = (˜ (a ^T))
  - => {a :} ((˜ a) ^-1) = (˜ (a ^-1))
  - == {a :Matrix, b :Matrix} ((a * b) ^T)=((b ^T) * (a ^T))
  - == {a :Matrix, b :Matrix} ((a + b) ^T)=((a ^T) + (b ^T))
  - == {a :Matrix, b :Matrix} ((a - b) ^T)=((a ^T) - (b ^T))
  - == {a :, b :, c :} ((a + b) * c)=((a * c) + (b * c))
  - == {a :, b :, c :} (a * (b + c))=((a * b) + (a * c))
  - == {a :, b :, c :} (a * (b - c))=((a * b) - (a * c))
  - == {a :, b :, c :} ((a - b) * c)=((a * c) - (b * c))
- Associativité :
  - == {a :, b :, c :} (a * (b * c))=((a * b) * c)
  - == {a :, b :, c :} (a + (b + c))=((a + b) + c)
- Autres :
  - == {a :, b :, c :} (a - (b - c))=((a - b) + c)
  - == {a :, b :, c :} (a + (b - c))=((a + b) - c)
  - == {a :, b :, c :} (a - (b + c))=((a - b) - c)
  - == {a :, b :} (a + (˜ b))=(a - b)
  - == {a :, b :} (a - (˜ b))=(a + b)
  - == {a :, b :} ((˜ a) * b)=(˜ (a * b))
  - => {a :SymDefPosMatrix} (((choleskyU a) ^T) * (choleskyU a))=a
  - => {a :SymDefPosMatrix} ((choleskyL a) * ((choleskyL a) ^T))=a
  - == {a :, b :, p :Vector} (((perm p a) ^-1) * b)=((a ^-1) * (perm p b))
  - => {a :} (perm (a pivlu) ((a l) * (a u)))=a
  - => {a :} (op ’n’ a)=a
  - => {a :} (op ’t’ a)=(a ^T)
  - => {a :InvMatrix} ((a ^-1) * a)=I
  - => {a :InvMatrix} (a * (a ^-1))=I
  - => {} (I ^-1)=I
  - => {a :SymMatrix} (a ^T)=a
  - => {} (I ^T)=I
  - => {} (O ^T)=O
  - => {a :} (-1 * a)=(˜ a)
  - => {a :} (-1.0 * a)=(˜ a)
  - => {a :} (I * a)=(˜ a)
  - => {} (˜ 1)=-1

- => {} (˜ 1.0)=-1.0
- => {} (˜ I)=-I
- => {} (˜ 0)=0
- => {} (˜ 0.0)=0.0
- => {} (˜ O)=O

B.4 Le BLAS

B.4.1 Niveau 1

- $sscal(n : Int, \alpha : Real, x : Vector, incx : Int)$:
  $x \leftarrow (\alpha * x)$
- $scopy(n : Int, x : Vector, incx : Int, y : Vector, incy : Int)$:
  $y \leftarrow x$
- $saxpy(n : Int, \alpha : Real, x : Vector, incx : Int, y : Vector, incy : Int)$:
  $y \leftarrow ((\alpha * x) + y)$
- $sdot(n : Int, x : Vector, incx : Int, y : Vector, incy : Int) : Real$
  $((x\hat{}T) * y)$
- $snrm2(n : Int, x : Vector, incx : Int) : Real$
  $norm2(x)$
- $sasum(n : Int, x : Vector, incx : Int) : Real$
  $norm1(x)$

B.4.2 Niveau 2

- $sgemv(trans : Char, m : Int, n : Int, \alpha : Real, a : Matrix, lda : Int, x : Vector, incx : Int, \beta : Real, y : Vector, incy : Int)$:
  $y \leftarrow match$
  $\mid (trans,' n') \rightarrow ((\alpha * (a * x)) + (\beta * y))$
  $\mid (trans,' t') \rightarrow ((\alpha * ((a\hat{}T) * x)) + (\beta * y))$
- $ssymv(uplo : Char, n : Int, \alpha : Real, a : Matrix, lda : Int, x : Vector, incx : Int, \beta : Real, y : Vector, incy : Vector)$:
  $y \leftarrow ((\alpha * (a * x)) + (\beta * y))$
- $strmv(uplo : Char, trans : Char, diag : Char, n : Int, a : Matrix, lda : Int, x : Vector, incx : Int)$:
  $[(a, UpTriMatrix, uplo,' u'), (a, LowTriMatrix, uplo,' l'),$
  $(a, UnitTriMatrix, diag,' u'), (a, TriMatrix, diag,' n')]$
  $x \leftarrow match$
  $\mid (trans,' n') \rightarrow (a * x)$
  $\mid (trans,' t') \rightarrow ((a\hat{}T) * x)$
- $strsv(uplo : Char, trans : Char, diag : Char, n : Int, a : TriInvMatrix, lda : Int, x : Vector, incx : Int)$:
  $[(a, UpTriInvMatrix, uplo,' u'), (a, LowTriInvMatrix, uplo,' l'),$
  $(a, UnitTriInvMatrix, diag,' u'), (a, TriInvMatrix, diag,' n')]$
  $a \leftarrow match$
  $\mid (trans,' n') \rightarrow ((a\hat{} - 1) * x)$
  $\mid (trans,' t') \rightarrow (((a\hat{} - 1)\hat{}T) * x)$

- $sger(m : Int, n : Int, \alpha : Real, x : Vector, incx : Int, y : Vector, incy : Int, a : Matrix, lda : Matrix) :$
  $a \leftarrow ((\alpha * (x * (y\hat{}T))) + a)$
- $ssyr(uplo : Char, n : Int, \alpha : Real, x : Vector, incx : Int, a : SymMatrix, lda : Matrix) :$
  $a \leftarrow ((\alpha * (x * (x\hat{}T))) + a)$
- $ssyr2(uplo : Char, n : Int, \alpha : Real, x : Vector, incx : Int, y : Vector, incy : Int, a : SymMatrix, lda : Int) :$
  $a \leftarrow (((\alpha * (x * (y\hat{}T))) + (\alpha * (y * (x\hat{}T)))) + a)$

B.4.3 Niveau 3

- $sgemm(transa : Char, transb : Char, m : Int, n : Int, k : Int, \alpha : Real, A : Matrix, lda : Int, B : Matrix, ldb : Int, \beta : Real, C : Matrix, ldc : Int) : Matrix$
  $C \leftarrow ((\alpha * (op(transa, A) * op(transb, B))) + (\beta * C))$
- $ssymm(side : Char, uplo : Char, m : Int, n : Int, \alpha : Real, A : SymMatrix, lda : Int, B : Matrix, ldb : Int, \beta : Real, C : Matrix, ldc : Int) : void$
  $C \leftarrow match$
  $\mid (side,' l') \rightarrow ((\alpha * (A * B)) + (\beta * C))$
  $\mid (side,' r') \rightarrow ((\alpha * (B * A)) + (\beta * C))$
- $ssyrk(uplo : Char, trans : Char, n : Int, k : Int, \alpha : Real, A : Matrix, lda : Int, \beta : Real, C : SymMatrix, ldc : Int) : void$
  $C \leftarrow match$
  $\mid (trans,' n') \rightarrow ((\alpha * ((A\hat{}T) * A)) + (\beta * C))$
  $\mid (trans,' t') \rightarrow ((\alpha * (A * (A\hat{}T))) + (\beta * C))$
- $ssyr2k(uplo : Char, trans : Char, n : Int, k : Int, \alpha : Real, A : Matrix, lda : Int, B : Matrix, ldb : Int, \beta : Real, C : SymMatrix, ldc : Int) : void$
  $C \leftarrow match$
  $\mid (trans,' n') \rightarrow (((\alpha * (A * (B\hat{}T))) + (\alpha * (B * (A\hat{}T)))) + (\beta * C))$
  $\mid (trans,' t') \rightarrow (((\alpha * ((A\hat{}T) * B)) + (\alpha * ((B\hat{}T) * A))) + (\beta * C))$
- $strmm(side : Char, uplo : Char, transA : Char, diag : Char, m : Int, n : Int, \alpha : Real, A : TriMatrix, lda : Int, B : Matrix, ldb : Int) : void$
  $B \leftarrow match$
  $\mid (side,' l') \rightarrow (\alpha * (op(transA, A) * B))$
  $\mid (side,' r') \rightarrow (\alpha * (B * op(transA, A)))$
- $strsm(side : Char, uplo : Char, transA : Char, diag : Char, m : Int, n : Int, \alpha : Real, A : TriInvMatrix, lda : Int, B : Matrix, ldb : Int) : void$
  $[(A, UpTriInvMatrix, uplo,' u'), (A, LowTriInvMatrix, uplo,' l'), (A, UnitTriInvMatrix, diag,' u'), (A, Matrix, diag,' n')]$
  $B \leftarrow match$
  $\mid (side,' l') \rightarrow (\alpha * (op(transA, (A\hat{} - 1)) * B))$
  $\mid (side,' r') \rightarrow (\alpha * (B * op(transA, (A\hat{} - 1))))$

Annexe C

Optimisation

C.1 Les types

- Matrix
- SymMatrix subsorte of Matrix
- Vector
- VectorL
- Real
- Int
- funVector->Vector
- funVector->Real subsorte of funVector->Vector
- FirstDerivFunVector->Real subsorte of funVector->Real
- SecDerivFunVector->Real subsorte of FirstDerivFunVector->Real
- funReal->Vector subsorte of funVector->Vector
- FirstDerivFunReal->Vector subsorte of funReal->Vector
- SecDerivFunReal->Vector subsorte of FirstDerivFunReal->Vector
- funReal->Real subsorte of funVector->Real funReal->Vector
- FirstDerivFunReal->Real subsorte of funReal->Real
- constraint

C.2 Les opérateurs

valmin et valmax représentent la valeur pour laquelle le minimum ou le maximum est atteint.

- min : funReal->Real $\times$ constraint$\rightarrow$ Real
- min : funVector->Real $\times$ constraint$\rightarrow$ Real
- min : funVector->Vector $\times$ constraint$\rightarrow$ Vector
- min : funReal->Vector $\times$ constraint$\rightarrow$ Vector
- valmin : funReal->Real $\times$ constraint$\rightarrow$ Real
- valmin : funReal->Vector $\times$ constraint$\rightarrow$ Real
- valmin : funVector->Real $\times$ constraint$\rightarrow$ Vector
- valmin : funVector->Vector $\times$ constraint$\rightarrow$ Vector
- max : funReal->Real $\times$ constraint$\rightarrow$ Real
- max : funVector->Real $\times$ constraint$\rightarrow$ Real
- max : funVector->Vector $\times$ constraint$\rightarrow$ Vector
- max : funReal->Vector $\times$ constraint$\rightarrow$ Vector

– valmax : funReal->Real × constraint→ Real
– valmax : funReal->Vector × constraint→ Real
– valmax : funVector->Real × constraint→ Vector
– valmax : funVector->Vector × constraint→ Vector
– solve : funVector->Vector→ Vector
– zero : funReal->Real→ Real
– & : constraint × constraint→ constraint [comm]
– -> : Real × Real→ funReal->Real
– -> : Real × Vector→ funReal->Vector
– -> : Vector × Real→ funVector->Real
– -> : Vector × Vector→ funVector->Vector
– < : Real × Real→ constraint
– < : Vector × Vector→ constraint
– <= : Real × Real→ constraint
– <= : Vector × Vector→ constraint
– = : Real × Real→ constraint [comm]
– = : Vector × Vector→ constraint [comm]
– * : Int × Int→ Int
– * : Real × Real→ Real
– * : Matrix × Vector→ Vector
– * : VectorL × Matrix→ VectorL
– * : Matrix × Matrix→ Matrix
– . : VectorL × Vector→ Real
– + : Int × Int→ Int [comm]
– + : Real × Real→ Real [comm]
– + : Vector × Vector→ Vector [comm]
– + : Matrix × Matrix→ Matrix [comm]
– - : Int × Int→ Int
– - : Real × Real→ Real
– - : Vector × Vector→ Vector
– - : Matrix × Matrix→ Matrix
– ~ : Int → Int
– ~ : Real → Real
– ~ : Vector → Vector
– ~ : Matrix → Matrix
– ^T : Vector→ VectorL
– n2 : Vector→ Real

– [ ] : → Matrix
– 1 : → Int
– -1 : → Int
– 0 : → Int
– 1.0 : → Real
– -1.0 : → Real
– 0.0 : → Real
– 1/2 : → Real
– I : → Matrix
– -I : → Matrix
– O : → Matrix

- xs : → Real
- [] : → Vector
- xv : → Vector
- Empty : → constraint

C.3 Les équations

- == {x :constraint, y :constraint, z :constraint} (x & (y & z))=((x & y) & z)
- => {x :Vector} (([ ] * x) <= [])=Empty
- => {x :} (([ ] * x) = [])=Empty
- => {a :constraint} (Empty & a)=a
- == {f :,c :constraint} (min(f,c))=(˜ (max((˜ f),c)))
- == {f :,c :constraint} (valmin(f,c))=(valmax((˜ f),c))

C.4 La boîte à outils Matlab

- $fminbnd(fun : funReal- > Vector, x1 : Real, x2 : Real) : Vector$
  $min(fun, ((x1 <= xs)\&(xs <= x2)))$
- $fminunc(fun : funVector- > Real, x0 : Vector) : Real$
  $min(fun, [])$
- $fzero(fun : funReal- > Real, x0 : Real) : Real$
  $zero(fun)$
- $fsolve(fun : funVector- > Vector, x0 : Vector) : Vector$
  $solve(fun)$
- $bintprog(f : Vector, A : Matrix, b : Vector, Aeq : Matrix, beq : Vector) :$
  $Vector$
  $min((xv- > ((f\hat{}T).xv)), (((A * xv) <= b)\&((Aeq * xv) = beq)))$
- $fmincon(fun : funVector- > Real, x0 : Vector, A : Matrix, b : Vector) :$
  $Real$
  $min(fun, ((A * xv) <= b))$
- $fmincon(fun : funVector- > Real, x0 : Vector, A : Matrix, b : Vector, Aeq :$
  $Matrix, beq : Vector) : Real$
  $min(fun, (((A * xv) <= b)\&((Aeq * xv) = beq)))$
- $fmincon(fun : funVector- > Real, x0 : Vector, A : Matrix, b : Vector, Aeq :$
  $Matrix, beq : Vector, lb : Vector, ub : Vector) : Real$
  $min(fun, ((((A * xv) <= b)\&((Aeq * xv) = beq))\&((lb <= xv)\&(xv <=$
  $ub))))$
- $linprog(f : Vector, A : Matrix, b : Vector) : Vector$
  $min((xv- > ((f\hat{}T).xv)), ((A * xv) <= b))$
- $linprog(f : Vector, A : Matrix, b : Vector, Aeq : Matrix, beq : Vector) :$
  $Vector$
  $min((xv- > ((f\hat{}T).xv)), (((A * xv) <= b)\&((Aeq * xv) = beq)))$
- $linprog(f : Vector, A : Matrix, b : Vector, Aeq : Matrix, beq : Vector, lb :$
  $Vector, ub : Vector) : Vector$
  $min((xv- > ((f\hat{}T).xv)), ((((A * xv) <= b)\&((Aeq * xv :) = beq))\&((lb <=$
  $xv)\&(xv <= ub))))$
- $lsqnonneg(C : Matrix, d : Vector) : Real$
  $min((xv- > (1/2 * n2(((C * xv) - d)))), (0. <= xv))$

- $lsqlin(C : Matrix, d : Vector, A : Matrix, b : Vector) : Real$
  $min((xv- > (1/2 * n2(((C * xv) - d)))), ((A * xv) <= b))$
- $lsqlin(C : Matrix, d : Vector, A : Matrix, b : Vector, Aeq : Matrix, beq : Vector) : Real$
  $min((xv- > (1/2*n2(((C*xv)-d)))), (((A*xv) <= b)\&((Aeq*xv) = beq)))$
- $lsqlin(C : Matrix, d : Vector, A : Matrix, b : Vector, Aeq : Matrix, beq : Vector, lb : Vector, ub : Vector) : Real$
  $min((xv- > (1/2 * n2(((C * xv) - d)))), ((((A * xv) <= b)\&((Aeq * xv) = beq))\&((lb <= xv)\&(xv <= ub))))$
- $quadprog(H : Matrix, f : Vector, A : Matrix, b : Vector) : Real$
  $min((xv- > ((1/2 * (((xv\hat{}T) * H).xv)) + ((f\hat{}T).xv))), ((A * xv) <= b))$
- $quadprog(H : Matrix, f : Vector, A : Matrix, b : Vector, Aeq : Matrix, beq : Vector) : Real$
  $min((xv- > ((1/2*(((xv\hat{}T)*H).xv))+((f\hat{}T).xv))), (((A*xv) <= b)\&((Aeq* xv) = beq)))$
- $quadprog(H : Matrix, f : Vector, A : Matrix, b : Vector, Aeq : Matrix, beq : Vector, lb : Vector, ub : Vector) : Real$
  $min((xv- > ((1/2 * (((xv\hat{}T) * H).xv)) + ((f\hat{}T).xv))), ((((A * xv) <= b)\&((Aeq * xvr) = beq))\&((lb <= xv)\&(xv <= ub))))$

C.5 E04 - NAG

- $E04ABF(FUNCT : funReal- > Real, E1 : Real, E2 : Real, A : Real, B : Real, MAXCAL : Int, X : Real, F : Real, IFAIL : Int) :$
  $X \leftarrow valmin(FUNCT, ((A <= xs)\&(xs <= B)))$
  $F \leftarrow min(FUNCT, ((A <= xs)\&(xs <= B)))$
- $E04DGF(N : Int, OBJFUN : funVector- > Real, ITER : Int, OBJF : Real, OBJGRD : Vector, X : Vector, IWORK : Vector, WORK : Vector, IUSER : Vector, USER : Vector, IFAIL : Int) :$
  $X \leftarrow valmin(OBJFUN, [])$
  $OBJF \leftarrow min(OBJFUN, [])$
- $E04JYF(N : Int, IBOUND : Int, FUNCT1 : funVector- > Vector, BL : Vector, BU : Vector, X : Vector, F : Real, IW : Vector, LIW : Int, W : Vector, LW : Int, IUSER : Vector, USER : Vector, IFAIL : Int) :$
  $X \leftarrow valmin(FUNCT1, ((BL <= xv)\&(xv <= BU)))$
  $F \leftarrow min(FUNCT1, ((BL <= xv)\&(xv <= BU)))$
- $E04BBF(FUNCT : FirstDerivFunReal- > Real, E1 : Real, E2 : Real, A : Real, B : Real, MAXCAL : Int, X : Real, F : Real, G : Real, IFAIL : Int) :$
  $X \leftarrow valmin(FUNCT, ((A <= xs)\&(xs <= B)))$
  $F \leftarrow min(FUNCT, ((A <= xs)\&(xs <= B)))$
- $E04MFF(N : Int, NCLIN : Int, A : Vector, LDA : Int, BL : Vector, BU : Vector, CVEC : Vector, ISTATE : Vector, X : Vector, ITER : Int, OBJ : Real, AX : Vector, CLAMDA : Vector, IWORK : Vector, LIWORK : Int, WORK : Vector, LWORK : Int, IFAIL : Int) :$
  $X \leftarrow valmin((xv- > ((CVEC\hat{}T).xv)), ((BL <= xv)\&(xv <= BU)))$
  $OBJ \leftarrow min((xv- > ((CVEC\hat{}T).xv)), ((BL <= xv)\&(xv <= BU)))$

- $E04NCF(M : Int, N : Int, NCLIN : Int, LDC : Int, LDA : Int, C : Vector, BL : Vector, BU : Vector, CVEC : Vector, ISTATE : Vector, KX : Vector, X : Vector, A : Matrix, B : Vector, ITER : Int, OBJ : Real, CLAMDA : Vector, IWORK : Vector, LIWORK : Int, WORK : Vector, LWORK : Int, IFAIL : Int) :$
  $X \leftarrow valmin((xv- > (1/2 * n2(((A * xv) - B)))), ((BL <= xv)\&(xv :<= BU)))$
  $OBJ \leftarrow min((xv- > (1/2 * n2(((A * xv) - B)))), ((BL <= xv)\&(xv <= BU)))$
- $E04NFF(N : Int, NCLIN : Int, A : Vector, LDA : Int, BL : Vector, BU : Vector, CVEC : Vector, H : SymMatrix, LDH : Int, QPHESS : funVector- > Vector, ISTATE : Vector, X : Vector, ITER : Int, OBJ : Real, AX : Vector, CLAMDA : Vector, IWORK : Vector, LIWORK : Int, WORK : Vector, LWORK : Int, IFAIL : Int) :$
  $X \leftarrow valmin((xv- > ((1/2*(((xv\hat{}T)*H).xv))+((CVEC\hat{}T).xv))), ((BL <= xv)\&(xv <= BU)))$
  $OBJ \leftarrow min((xv- > ((1/2*(((xv\hat{}T)*H).xv))+((CVEC\hat{}T).xv))), ((BL <= xv)\&(xv <= BU)))$

Bibliographie

[AAB+01] Dorian Arnold, Sudesh Agrawal, Susan Blackford, Jack Dongarra, Michelle Miller, Kiran Sagi, Zhiao Shi, and Sathish Vadhiyar. Users' Guide to NetSolve V1.4. Computer Science Dept. Technical Report CS-01-467, University of Tennessee, Knoxville, TN, July 2001.

[ABB+99] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, Jim Demmel, Jack Dongarra, Jeremey Du Croz, Sven Hammarling, Anne Greenbaum, Alan McKenney, and Danny Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[ABJ05] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. Juxmem : An adaptive supportive platform for data sharing on the grid. *Scalable Computing : Practice and Experience*, 6(3) :45–55, September 2005.

[ABK+02] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL : The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2) :153–196, September 2002.

[ABMN02] Olivier Aumage, Luc Bougé, Jean-François Méhaut, and Raymond Namyst. Madeleine ii : A portable and efficient communication library for high-performance cluster computing. *Parallel Computing*, 28(4) :607–626, April 2002.

[Abr96] Jean-Raymond Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

[AEH94] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. In *POPL '94 : Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 268–279, New York, NY, USA, 1994. ACM Press.

[AGM97] Peter Arbenz, Walter Gander, and Junichiro Mori. The Remote Computational System. *Parallel Computing*, 23(10) :1421–1428, 1997.

[AT85] Stefan Arnborg and Erik Tidén. Unification problems with one-sided distributivity. In *Proc. of the first international conference on Rewriting techniques and applications*, pages 398–406, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[AvH03] Grigoris Antoniou and Frank van Harmelen. Web ontology language : Owl. In S. Staab and R. Studer, editors, *Handbook on Ontologies in Information Systems*. Springer-Verlag, 2003.

[BC85] Michel Bidoit and Christine Choppy. Asspegique : an integrated environment for algebraic specifications. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT) on Formal Methods and Software, Vol.2 : Colloquium on Software Engineering (CSE)*, pages 246–260, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[BCW88] Richard Becker, John Chambers, and Allan Wilks. *The new S language : a programming environment for data analysis and graphics*. Wadsworth and Brooks/Cole Advanced Books & Software, Monterey, CA, USA, 1988.

[BDD+02] Susan Blackford, Jim Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Mike Heroux, Linda Kaufman, A. Lumsdaine, Andrew Petitet, Roldan Pozo, Karin Remington, and Clint Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.*, 28(2) :135–151, 2002.

[BE86] Didier Bert and Rachid Echahed. Design and implementation of a generic, logic and functional programming language. In *ESOP '86 : Proceedings of the European Symposium on Programming*, pages 119–132, London, UK, 1986. Springer-Verlag.

[BKK+98] Peter Borovansky, Claude Kirchner, Helène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of elan, 1998.

[BM04] Michel Bidoit and Peter D. Mosses. CASL *User Manual*. LNCS 2900 (IFIP Series). Springer, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, United Kingdom, 1998.

[BS96] Franz Baader and Klaus Schulz. Unification in the union of disjoint equational theories : combining decision procedures. *Journal of Symbolic Computation*, 21(2) :211–243, 1996.

[BS01] Franz Baader and Wayne Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.

[Bür89] Hans-Jürgen Bürckert. Matching - a special case of unification ? *J. Symb. Comput.*, 8(5) :523–536, 1989.

[CDD+05] Eddy Caron, Frédéric Desprez, Michel Daydé, Aurélie Hurault, and Marc Pantel. On deploying scientific software within the grid-tlse project. *Computing Letters (CoLe)*, 1(3) :1–5, 2005.

[CDE+00] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. *A Maude Tutorial*. SRI International, 2000.

[CDE+03] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The maude 2.0 system. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer-Verlag, June 2003.

[CDL+02] Eddy Caron, Frédéric Desprez, Frédéric Lombard, Jean-Marc Nicod, Martin Quinson, and Frédéric Suter. A scalable approach to network enabled servers.

In B. Monien and R. Feldmann, editors, *Proceedings of the 8th International EuroPar Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 907–910, Paderborn, Germany, August 2002. Springer-Verlag.

[CDL+06] Eddy Caron, Frédéric Desprez, Jean-Yves L'Excellent, Christophe Hamerling, Marc Pantel, and Chiara Puglisi-Amestoy. Future Generation Grids. volume XVIII, CoreGrid Series of *Proceedings of the Workshop on Future Generation Grids November 1-5, 2004, Dagstuhl, Germany*, chapter Use of A Network Enabled Server System for a Sparse Linear Algebra Application. Springer Verlag, 2006.

[CEW93] Ingo Classen, Hartmut Ehrig, and Dietmar Wolz. *Algebraic specification techniques and tools for software development : the ACT approach*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1993.

[CGL92] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, volume 5, pages 182–192, 1992.

[CMU04] Evelyne Contejean, Claude Marché, and Xavier Urbain. CiME3, 2004. Available at `http://cime.lri.fr/`.

[Con04] Evelyne Contejean. A certified AC matching algorithm. In Vincent van Oostrom, editor, *15th International Conference on Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, pages 70–84, Aachen, Germany, June 2004. Springer-Verlag.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4) :471–522, 1985.

[DDDH90] Jack Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. Algorithm 679. a set of Level 3 Basic Linear Algebra S ubprograms. *ACM Transactions on Mathematical Software*, 16 :1–17, 1990.

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17 :279–301, 1982.

[DGG+94] Luiz DeRose, Kyle Gallivan, Efstratios Gallopoulos, Bret Marsolf, and David Padua. An environment for the rapid prototyping and development of numerical programs and libraries for scientific computation, 1994.

[DGH+04] Michel Daydé, Luc Giraud, Montse Hernandez, Jean-Yves L'Excellent, Marc Pantel, and Chiara Puglisi. An Overview of the GRID-TLSE Project . In Michel Daydé, Jack Dongarra, Vicente Hernandez, and Jose Palma, editors, *6th International Meeting VECPAR'04 , Valencia, Espagne, 28/06/04-30/06/04*, pages 851–856. Universidad Politécnica de Valencia, juin 2004.

[DHP05] Michel Daydé, Aurélie Hurault, and Marc Pantel. Gridification of scientific application using software components : The grid-tlse project as an illustration. In *CSIT, Fifth International Conference on Computer Science and Information Technologies, Yerevan, Armenia*, pages 419–427, 9-23 September 2005.

[DHP06] Michel Daydé, Aurélie Hurault, and Marc Pantel. Semantic-based service trading : Application to linear algebra. In *VECPAR'2006, Rio de Janeiro, Brazil*, 10-13 July 2006. To appear.

[DJ90] Daniel Dougherty and Patricia Johann. An improved general e-unification method. In *Conference on Automated Deduction*, pages 261–275, 1990.

[DMS92] Nachum Dershowitz, Subrata Mitra, and G. Sivakumar. Decidable matching for convergent systems. In D. Kapur, editor, *Proceedings of the Eleventh Conference on Automated Deduction (Saratoga Springs, NY)*, volume 607, pages 589–602, Berlin, 1992. Springer-Verlag.

[Dom91] Eric Domenjoud. *Outils pour la déduction automatique dans les théories associatives-commutatives*. PhD thesis, Université Nancy I, 1991.

[DPPR04] Alexandre Denis, Christian Pérez, Thierry Priol, and André Ribes. Bringing high performance to the corba component model. In *SIAM Conference on Parallel Processing for Scientific Computing*, February 2004.

[DPR05] Roberto Di Cosmo, François Pottier, and Didier Rémy. Subtyping recursive types modulo associative commutative products. In *Seventh International Conference on Typed Lambda Calculi and Applications (TLCA'05)*, Nara, Japan, April 2005.

[FGNC01] Gilles Fedak, Cécile Germain, Vincent Neri, and Franl Cappello. XtremWeb : A Generic Global Computing System. In *CCGRID '01 : Proceedings of the 1st International Symposium on Cluster Computing and the Grid*. IEEE Computer Society, 2001.

[FH86] François Fages and Gérard Huet. Complete Sets of Unifiers and Matchers in Equational Theories. *Theoretical Computer Science*, 43(2-3) :189–200, 1986.

[FK97] Ian Foster and Carl Kesselman. Globus : A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2) :115–128, Summer 1997.

[FLM+05] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[FSYH03] Yoshinari Fukui, Andrew Stubbings, Takashi Yamazaki, and Ryutaro Himeno. Constructing a virtual laboratory on the internet : The itbl portal. In Alexander V. Veidenbaum, Kazuki Joe, Hideharu Amano, and Hideo Aiso, editors, *ISHPC*, volume 2858 of *Lecture Notes in Computer Science*, pages 288–297. Springer, 2003.

[GGHv01] John Gunnels, Fred Gustavson, Greg Henry, and Robert van de Geijn. FLAME : Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4) :422–455, December 2001.

[GGM99] Jean-Marie Geib, Christophe Gransart, and Philippe Merle. *CORBA : des concepts à la pratique*. DUNOD, 1999.

[GH86] John Guttag and James Horning. Report on the larch shared language. *Sci. Comput. Program.*, 6(2) :103–134, 1986.

[GM92] Joseph Goguen and José Meseguer. Order-sorted algebra i : equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theor. Comput. Sci.*, 105(2) :217–273, 1992.

[Gor88] M. Gordon. HOL : A Proof Generating System for Higher-order Logic. In G. Birwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–127. Academic Press, Boston, 1988.

[GS89] Jean Gallier and Wayne Snyder. Complete Sets of Transformations for General E-Unification. *Theor. Comput. Sci.*, 67(2-3) :203–260, 1989.

[GvH00] John Gunnels, Robert van de Geijn, and Greg Henry. Formal linear algebra methods environment (FLAME) overview, November 2000.

[GWM+93] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.

[Han94] Michael Hanus. Lazy unification with simplification. In *Proc. 5th European Symposium on Programming (Edinburgh, Scotland)*, volume 788, pages 272–286, Berlin, Germany, 1994. Springer.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, 1969.

[Hoa83] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 26(1) :53–56, 1983.

[HP05] Aurélie Hurault and Marc Pantel. Mathematical service trading based on equational matching. Technical Report TR/TLSE/05/04, ENSEEIHT-IRIT, 2005.

[HP06] Aurélie Hurault and Marc Pantel. Mathematical service trading based on equational matching. In *Proceedings of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus 2005)*, volume 151, pages 161–177. Electronic Notes in Theoretical Computer Science, 21 March 2006.

[HPD04] Aurélie Hurault, Marc Pantel, and Frédéric Desprez. Recherche de services en algèbre linéaire sur une grille. In *RenPar'16*, 2004.

[HR04] Thérèse Hardin and Renaud Rioboo. Les objets des mathématiques. *RSTI - L'objet*, Octobre 2004.

[Hul80] Jean-Marie Hullot. Canonical forms and unification. In *Proceedings of the 5th Conference on Automated Deduction*, pages 318–334, London, UK, 1980. Springer-Verlag.

[Jaf90] Joxan Jaffar. Minimal and complete word unification. *J. ACM*, 37(1) :47–85, 1990.

[KB70] Donald Knuth and Peter Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Word Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[KBC+01] Ken Kennedy, Bradley Broom, Keith Cooper, Jack Dongarra, Rob Fowler, Dennis Gannon, Lennart Johnsson, John Mellor-Crummey, and Linda Torczon. Telescoping languages : A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12) :1803&ndash ;1826, 2001.

[KBC+05] Ken Kennedy, Bradley Broom, Arun Chauhan, Rob Fowler, John Garvin, Charles Koelbel, Cheryl McCosh, and John Mellor-Crummey. Telescoping languages : A system for automatic generation of domain languages. *Proceedings of the IEEE*, Volume 93(3) :387˘2013408, 2005. This provides a current overview of the entire Telescoping Languages Project.

[KFNM04] Holger Knublauch, Ray W. Fergerson, Natalya F. Noy, and Mark A. Musen. The protégé owl plugin : An open development environment for semantic web applications. In *Third International Semantic Web Conference - ISWC 2004*, Hiroshima, Japan, 2004.

[Kir88] Claude Kirchner. Oerder-sorted equational unification. Technical report, INRIA-Lorraine, 1988.

[Lal90] René Lalement. *Logique, réduction, résolution*. 1990.

[LS99] Ora Lassila and Ralph Swick. Resource Description Framework (RDF) Model and Syntax Specification, 1999.

[Mak77] Gennadi Semyonovich Makanin. The problem of solvability of equations in a free semi-group. *Math. USSR Sbornik*, 32(2) :129–198, 1977.

[MD96] Subrata Mitra and Nachum Dershowitz. Matching and unification in rewrite theories, 1996.

[MGS89] José Meseguer, Joseph Goguen, and Gert Smolka. Order-sorted unification. *J. Symb. Comput.*, 8(4) :383–413, 1989.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2) :258–282, 1982.

[MMM98] Ali Mili, Rym Mili, and Roland Mittermeir. A survey of software reuse libraries. *Ann. Softw. Eng.*, 5 :349–414, 1998.

[MNSS00] Satoshi Matsuoka, Hidemoto Nakada, Mitsuhisa Sato, and Satoshi Sekiguchi. Design Issues of Network Enabled Server Systems for the Grid. `http://www.eece.unm.edu/~dbader/grid/WhitePapers/satoshi.pdf`, 2000. Grid Forum, Advanced Programming Models Working Group whitepaper.

[MvH04] Deborah McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview, W3C Recommendation, 2004.

[NPW02] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[NSD+01] Natalya F. Noy, Michael Sintek, Stefan Decker, Monica Crubezy, Ray W. Fergerson, and Mark A. Musen. Creating semantic web contents with protege-2000. *IEEE Intelligent Systems*, 2(16) :60–71, 2001.

[ORSSC98] Sam Owre, John Rushby, Natarajan Shankar, and David Stringer-Calvert. PVS : an experience report. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullman, editors, *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Boppard, Germany, oct 1998. Springer-Verlag.

[PD02] Virgile Prevosto and Damien Doligez. Algorithms and proof inheritance in the foc language. *Journal of Automated Reasoning*, 29(3-4) :337–363, December 2002.

[PW76] Mike Paterson and Mark Wegman. Linear unification. In *STOC '76 : Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 181–186, New York, NY, USA, 1976. ACM Press.

[R69] George Robinson and Lawrence Wos . Paramodulation and theorem-proving in first-order theories with equality. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 4*, pages 135–150. Edinburgh University Press, Edinburgh, Scotland, 1969.

[Rit89] Mikael Rittri. Using types as search keys in function libraries. In *FPCA '89 : Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 174–183, New York, NY, USA, 1989. ACM Press.

[RT89] Colin Runciman and Ian Toyn. Retrieving reusable software components by polymorphic type. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 166–173, London, UK, 1989. ACM Press.

[SA94] Rolf Socher-Ambrosius. A refined version of general e-unification. In *Conference on Automated Deduction*, pages 665–677, 1994.

[Sch92] Klaus Schulz. Makanin's algorithm for word equations - two improvements and a generalization. In *IWWERT '90 : Proceedings of the First International Workshop on Word Equations and Related Topics*, pages 85–150, London, UK, 1992. Springer-Verlag.

[Sie79] Jörg Siekmann. Unification of commutative terms. In *EUROSAM '79 : Proceedings of the International Symposiumon on Symbolic and Algebraic Computation*, page 22, London, UK, 1979. Springer-Verlag.

[SLD+04] Keith Seymour, Craig Lee, Frédéric Desprez, Hidemoto Nakada, and Yoshio Tanaka. The End-User and Middleware APIs for GridRPC. In *Workshop on Grid Application Programming Interfaces, In conjunction with GGF12*, Brussels, Belgium, September 2004.

[SNGM89] Gert Smolka, Werner Nutt, Joseph Goguen, and José Meseguer. Order-sorted equational computation. In Maurice Nivat and Hassan Aït-Kaci, editors, *Resolution of Equations in Algebraic Structures*, volume 2, pages 297–367. Academic Press, 1989.

[SS96] Manfred Schmidt-Schauß. Decidability of unification in the theory of one-sided distributivity and a multiplicative unit. *J. Symb. Comput.*, 22(3) :315–344, 1996.

[SW83] Donald Sannella and Martin Wirsing. A kernel language for algebraic specification and implementation - extended abstract. In *Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory*, pages 413–427, London, UK, 1983. Springer-Verlag.

[SWL+94] Mark Stickel, Richard Waldinger, Michael Lowry, Thomas Pressburger, and Ian Underwood. Deductive composition of astronomical software from subroutine libraries. In *Conference on Automated Deduction*, pages 341–355, 1994.

[The04] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, April 2004. `http://coq.inria.fr`.

[TNS+03] Yoshio Tanaka, Hidemoto Nakada, Satoshi Sekiguchi, Toyotaro Suzumura, and Satoshi Matsuoka. Ninf-G : A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1(1) :41–51, 2003.

[TTL05] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice : the condor experience. *Concurrency - Practice and Experience*, 17(2-4) :323–356, 2005.

[vDM96] Arie van Deursen and Peter D. Mosses. ASD : The action semantic description tools. In *AMAST'96, Proc. 5th Intl. Conf. on Algebraic Methodology and Software Technology, Munich*, volume 1101, pages 579–582. Springer-Verlag, 1996.

[ZW93] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching : A key to reuse. In David Notkin, editor, *Proceedings of theFirstACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 182–190. ACM Press, 1993.